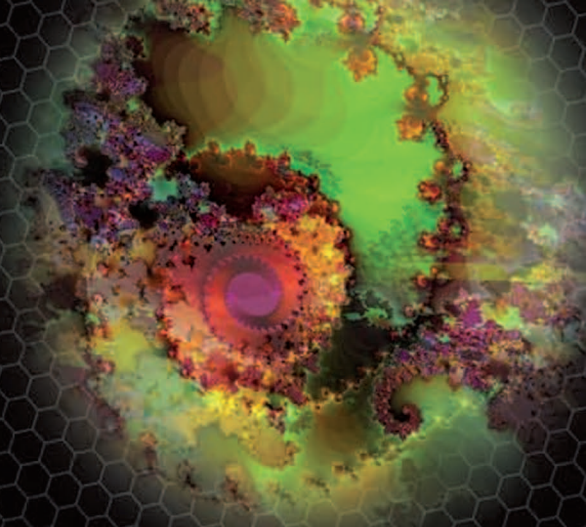


La guia definitiva de

django

Desarrolla aplicaciones Web de forma rápida y sencilla.

Django es un framework que ahorra tu tiempo
y hace del desarrollo Web una diversión

Adrian Holovaty
y **Jacob Kaplan Mosss**
Benevolentes dictadores de Django

Saul Garcia M.

Django Software Corporation

La guía definitiva de Django: Desarrolla aplicaciones web de forma rápida y sencilla.

Este libro ha sido preparado utilizando el lenguaje de marcas RST y la maquinaria de LATEX , para formatear el texto, para editar los gráficos se utilizo Gimp y para el maquetado Scribus , las imágenes se obtuvieron libremente de OpenClipart .

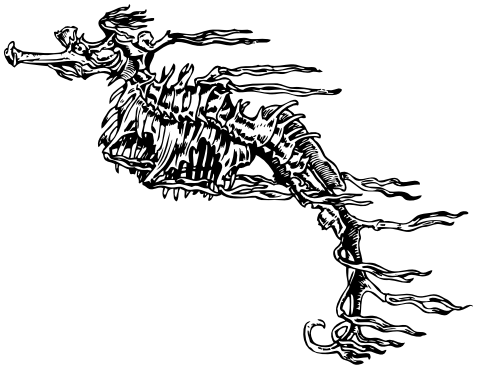
Todos estos programas son de código abierto y gratuitos.

Ideas, sugerencias; quejas: saulgarciamonroy@gmail.com

Publicado, editado y compilado: en algún lugar de Celayita México.

Django, la guía definitiva

Desarrolla aplicaciones Web de forma rápida y sencilla usando Django



Saul Garcia M.

Django Software Corporation

Capítulos

PARTE 4 Apéndices de referencia

Sobre los autores

ADRIAN HOLOVATY, es un periodista y desarrollador Web, conocido en los círculos editoriales, como uno de los pioneros en el uso de la programación en el periodismo, también es conocido en los círculos técnicos como el "Tipo que invento Django".

Adrian trabajo como desarrollador en World Online cerca de 2.5 años, tiempo en el cual desarrollo Django e implemento algunos sitios en World Online's. Es el fundador de EveryBlock, una Web startup local de noticias y de Soundslice.

Adrian vive in Chicago, USA.

JACOB KAPLAN-MOSS, es socio de Revolution Systems la cual provee soporte y servicios de asistencia relacionadas con tecnologias libres alrededor de Django.

Jacob trabajo anteriormente en World Online donde fue desarrollado Django, actualmente supervisa el desarrollo de Ellington, una plataforma de publicación online de noticias para compañías de medios de comunicación.

Jacob vive en Lawrence, Kansas, USA.

Sobre el editor técnico

JEREMY DUNCK: fue rescatado del aburrido trabajo corporativo por el software libre, en parte por Django. Muchos de los intereses de Jeremy se centran en torno al acceso a la información.

Es el principal desarrollador de Pegasus News, uno de los primeros sitios que usaron Django fuera de World Online, desde entonces se unio a Votizen, un startup que intenta reducir la influencia del dinero en la política.

Sirve como secretario en DSF, organiza y ayuda a organizar eventos, cuida la vitalidad y la equidad en la comunidad Django. Aunque lleva bastante tiempo sin escribir en un blog.

Jeremy vive en Mountain View, CA. USA.

Sobre este libro

Estás leyendo *El libro de Django*, libro publicado por primera vez en Diciembre de 2007 por la editorial Apress con el título de: The Definitive Guide to Django: Web Development Done Right.

Hemos lanzado esta versión libremente, por un par de razones:

- La primera es que amamos Django y queremos que sea tan accesible como sea posible. Muchos programadores aprenden su arte, usando material técnico bien escrito, así que nosotros intentamos escribir una guía destacada, que sirva además como referencia para usar Django.

- La segunda, es que resulta que escribir libros sobre tecnología es particularmente difícil: sus palabras se vuelven anticuadas incluso antes de que el libro llegue a la imprenta. En la Web, sin embargo, la tinta nunca se seca por lo que podremos mantener este libro al día (y así lo haremos).

La respuesta de los lectores es una parte crítica de ese proceso. Hemos construido un sistema de comentarios que te dejará comentar sobre cualquier parte del libro; leeremos y utilizaremos estos comentarios en nuevas versiones.

Sobre esta versión

La traducción al español de "El libro de Django" fue posible gracias a la colaboración voluntaria de la comunidad Django en español.

Este libro fue actualizado en 2009 y cubría Django 1.1. Desde entonces ha quedado un poco desactualizado, es por ello que estamos trabajando en la actualización del libro, para que cubra Django 2.0. Sin embargo necesitamos de tu ayuda para lograrlo. Es por ello que decidimos compartir este libro, con la esperanza de que encuentres en él un proyecto comunitario libre y en constante evolución.

Así que si quieres echarnos una mano, ¡toda ayuda será bien recibida!

El código original de este libro, en la versión en Ingles está alojado en GitHub en http://github.com/jacobian/djangobook.com, mientras que la versión en español está alojada en http://github.com/saulgm/djangobook.com, en proceso de actualización.

¡Toda ayuda será bien recibida!

Compilador:	**Saul García M.**
Titulo:	Django, la guía definitiva
Versión:	1.8.0
Ultima actualización:	13 de Enero de 2015

A la fecha, han contribuido de una u otra manera a este trabajo: Manuel Kaufmann, Martín Gaitán, Leonardo Gastón De Luca, Guillermo Heizenreder, Alejandro Autalán Renzo Carbonara, Milton Mazzarri, Ramiro Morales, Juan Ignacio Rodríguez de León, Percy Pérez Pinedo, Tomás Casquero, Marcos Agustín Lewis, Leónidas Hernán Olivera, Federico M. Peretti, César Ballardini, Anthony Lenton, César Roldán, Gonzalo Delgado.

Reconocimientos

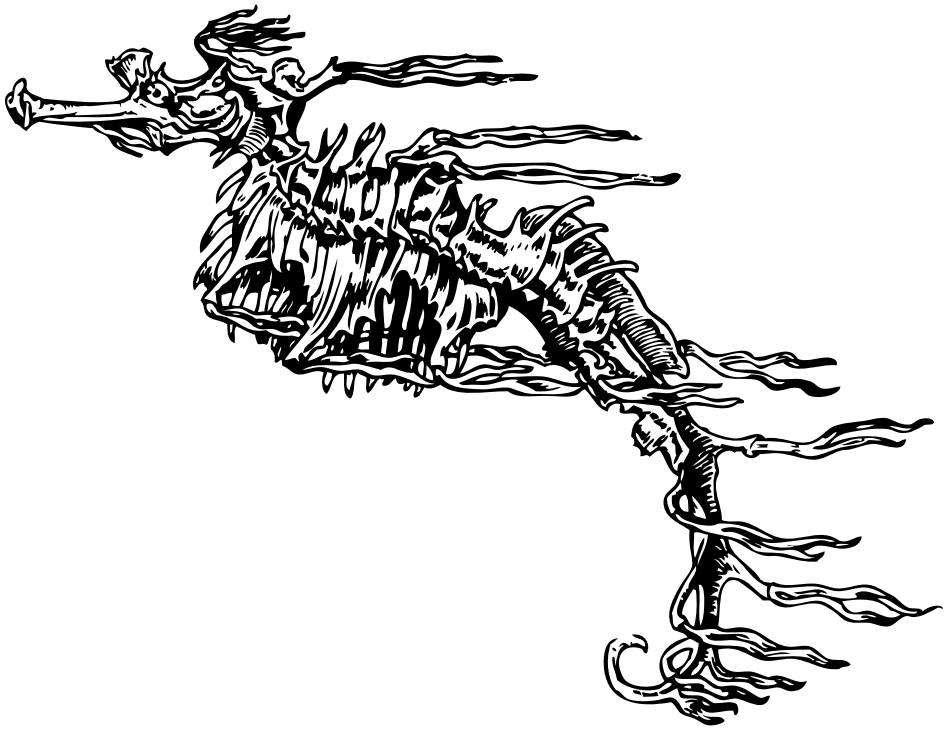
El aspecto más gratificante al trabajar con Django es la comunidad. Hemos sido especialmente afortunados que Django haya atraído a tanta gente inteligente, motivada y amistosa. Una buena parte de esa comunidad nos siguió durante el lanzamiento online beta de este libro. Sus revisiones y comentarios fueron indispensables; este libro no hubiese sido posible sin esa maravillosa revisión de pares. Casi mil personas dejaron comentarios que ayudaron a mejorar la claridad, calidad y el flujo del libro final. Queremos agradecer a todos y cada uno de ellos.

Estamos especialmente agradecidos con aquellos que dispusieron de su tiempo para revisar el libro en profundidad y dejarnos decenas (a veces cientos) de comentarios: Marty Alchin, Max Battcher, Oliver Beat-tie, Rod Begbie, Paul Bissex, Matt Boersma, RobbinBonthond, Peter Bowyer, Nesta Campbell, Jon Colverson, Jeff Croft, Chris Dary,Alex Dong, Matt Drew, Robert Dzikowski, Nick Efford, Ludvig Ericson, Eric Floehr, Brad Fults, David Grant, Simon Greenhill, Robert Haveman, Kent Johnson, Andrew Kember, Marek Kubica, Eduard Kucera, Anand Kumria, Scott Lamb, Fredrik Lundh, Vadim Macagon, Markus Majer, Orestis Markou, R. Mason, Yasushi Masuda, Kevin Menard, Carlo Miron, James Mulholland, R.D. Nielsen, Michael O'Keefe, Lawrence Oluyede, Andreas Pfrengle, Frankie Robertson, Mike Robinson, Armin Ronacher, Daniel Roseman, Johan Samyn, Ross Shannon, Carolina F. Silva, Paul Smith, Björn Stabell, Bob Stepno, Graeme Stevenson, Justin Stockton, Kevin Teague, Daniel Tietze, Brooks Travis, Peter Tripp, Matthias Urlichs, Peter van Kampen, Alexandre Vassalotti, Jay Wang, Brian Will, y Joshua Works.

Muchas gracias a nuestro editor técnico, Jeremy Dunck. Sin Jeremy, este libro habría quedado en desorden, con errores, inexactitudes y código roto. Nos sentimos realmente afortunados de que alguien con el talento de Jeremy encontrase el tiempo para ayudarnos.

Estamos agradecidos por todo el duro trabajo que la gente de Apress hizo en este libro. Su ayuda y paciencia ha sido asombrosa; este libro no habría quedado terminado sin todo ese trabajo de su parte. Nos pone especialmente felices que Apress haya apoyado e incluso alentado el lanzamiento libre de este libro online; es maravilloso ver a un editor tan abrazado al espíritu del open source.

Finalmente, por supuesto, gracias a nuestros amigos, familias y compañeros que gentilmente toleraron nuestra ausencia mental mientras terminábamos este trabajo.



Introducción

Al comienzo de internet, los desarrolladores Web escribían cada una de las páginas a mano. Actualizar un sitio significaba editar HTML; un "rediseño" implicaba rehacer cada una de las páginas, una a la vez.

Como los sitios Web crecieron y se hicieron más ambiciosos, rápidamente se hizo evidente que esta situación era tediosa, consumía tiempo y al final era insostenible. Un grupo de emprendedores del *NCSA* (Centro Nacional de Aplicaciones para Supercomputadoras, donde se desarrollo el Mosaic; el primer navegador Web gráfico) solucionó este problema permitiendo que el servidor Web invocara programas externos capaces de generar HTML dinámicamente. Ellos llamaron a este protocolo "Puerta de Enlace Común", o CGI[1], y esto cambió internet para siempre.

Ahora es difícil imaginar la revelación que CGI debe haber sido: en vez de tratar con páginas HTML como simples archivos del disco, CGI te permite pensar en páginas como recursos generados dinámicamente bajo demanda. El desarrollo de CGI hace pensar en la primera generación de página Web dinámicas.

Sin embargo, CGI tiene sus problemas: los scripts CGI necesitan contener gran cantidad de código repetitivo que los hace difícil de reutilizar, así como complicados de entender y escribir para los desarrolladores novatos.

PHP solucionó varios de estos problemas y tomó al mundo por sorpresa –ahora es, por lejos, la herramienta más popular usada para crear sitios Web dinámicos, y decenas de lenguajes y entornos similares (ASP, JSP, etc.) siguieron de cerca el diseño de PHP. La mayor innovación de PHP es que es fácil de usar: el código PHP es simple de embeber en un HTML plano; la curva de aprendizaje para algunos que recién conocen HTML es extremadamente llana.

Pero PHP tiene sus propios problemas; por su facilidad de uso, alienta a la producción de código mal hecho. Lo que es peor, PHP hace poco para proteger a los programadores en cuanto a vulnerabilidades de seguridad, por lo que muchos desarrolladores de PHP se encontraron con que tenían que aprender sobre seguridad cuando ya era demasiado tarde.

Estas y otras frustraciones similares, condujeron directamente al desarrollo de los actuales frameworks de desarrollo Web de **"tercera generación"**. Estos frameworks Django y Ruby on Rails – parecen ser muy populares en estos días – reconocen que la importancia de la Web se ha intensificado en los últimos tiempos. Con esta nueva explosión del desarrollo Web comienza otro incremento en la ambición; se espera que los desarrolladores Web hagan más y más cada día.

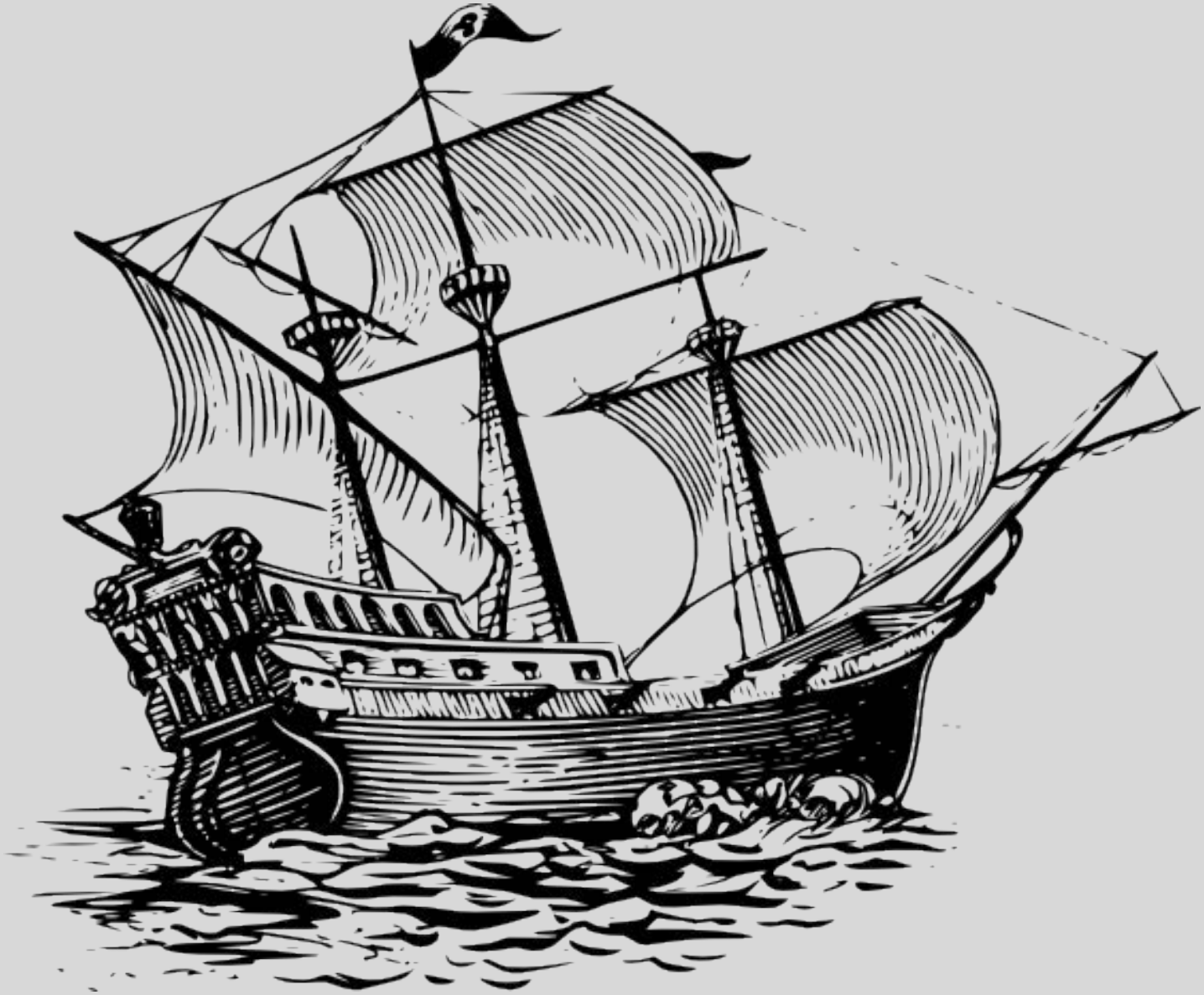
[1] N. del T.: Common Gateway Interface

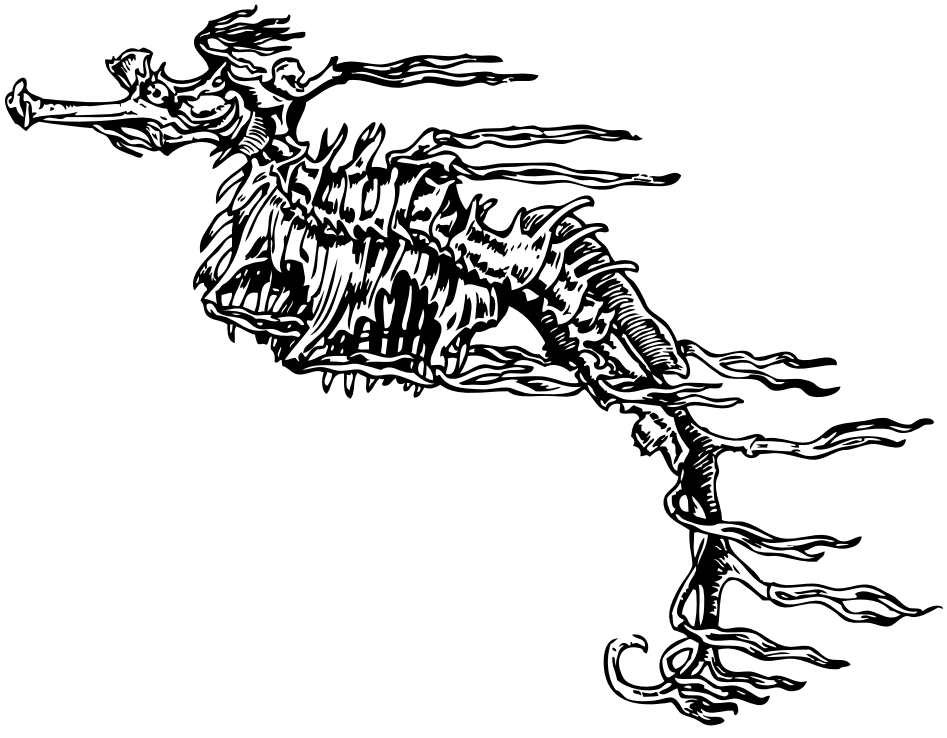
Django fue desarrollado para satisfacer esas nuevas ambiciones. Django te permite construir en profundidad, de forma dinámica, sitios interesantes en un tiempo extremadamente corto. Django está diseñado para hacer foco en la diversión, en las partes interesantes de tu trabajo, al mismo tiempo que alivia el dolor de las partes repetitivas. Al hacerlo, proporciona abstracciones de alto nivel a patrones comunes del desarrollo Web, agrega atajos para tareas frecuentes de programación y claras convenciones sobre cómo resolver problemas. Al mismo tiempo, intenta mantenerse fuera de tu camino, dejando que trabajes fuera del alcance del framework cuando sea necesario. Escribimos este libro porque creemos firmemente que Django mejora el desarrollo Web. Está diseñado para poner rápidamente en movimiento tu propio proyecto de Django, en última instancia aprenderás todo lo que necesites saber para producir el diseño, desarrollo y despliegue de sitios satisfactorios y de los cuales te sientas orgulloso.

Estamos extremadamente interesados en la retroalimentación. La versión online de este libro te permite dejar un comentario en cualquier parte del libro y discutir con otros lectores. Hacemos cuanto podemos para leer todos los comentarios posteados allí y responder tantos como nos sea posible. Si prefieres utilizar correo electrónico, por favor envíanos unas líneas (en inglés) a feedback@djangobook.com. De cualquier modo, ¡nos encantaría escucharte! Nos alegra que estés aquí, y esperamos que encuentres a Django tan emocionante, divertido y útil como nosotros.

PARTE 1

Comenzando





CAPÍTULO 1

Introducción a Django

Este libro es sobre **Django**, un framework de desarrollo Web que ahorra tiempo y hace que el desarrollo Web sea divertido. Utilizando Django puedes crear y mantener aplicaciones Web de alta calidad con un mínimo esfuerzo.

En el mejor de los casos, el desarrollo Web es un acto entretenido y creativo; en el peor, puede ser una molestia repetitiva y frustrante. Django te permite enfocarte en la parte creativa – la parte divertida de tus aplicaciones Web al mismo tiempo que mitiga el esfuerzo de las partes repetitivas. De esta forma, provee un alto nivel de abstracción de patrones comunes en el desarrollo Web, atajos para tareas frecuentes de programación y convenciones claras sobre cómo solucionar problemas. Al mismo tiempo, Django intenta no entrometerse, dejándote trabajar fuera del ámbito del framework según sea necesario.

El objetivo de este libro es convertirte en un experto de Django. Por lo que el enfoque es doble, primero, explicamos en profundidad lo que hace Django, y cómo crear aplicaciones Web con él, segundo, discutiremos conceptos de alto nivel cuando se considere apropiado, contestando la pregunta ¿Cómo puedo aplicar estas herramientas de forma efectiva en mis propios proyectos? Al leer este libro, aprenderás las habilidades necesarias para desarrollar sitios Web conectados a una base de datos, poderosos, de forma rápida, con código limpio y de fácil mantenimiento.

En este capítulo presentamos una visión general sobre Django.

¿Qué es un Framework Web?

Django es un miembro importante de una nueva generación de **frameworks Web**. ¿Pero qué significa este término exactamente?

Para contestar esa pregunta, consideremos el diseño de una aplicación Web escrita en Python, *sin usar* un framework. Una de las formas más simples y directas para construir una aplicación Web desde cero en python, es usando el estándar Common Gateway Interface (CGI), una técnica muy popular para escribir aplicaciones Web alrededor del año 1998. Esta es una explicación de alto nivel sobre cómo trabaja. Solo crea un script Python, que produzca HTML, guarda el script en el servidor Web con la extensión .cgi y visita la pagina con un navegador Web. Eso ¡Eso todo!

Por ejemplo, aquí hay un sencillo script CGI, escrito en python 2.x, que muestra los diez últimos libros publicados en una base de datos. No te preocupes por los detalles de la sintaxis; solo observa las cosas básicas que hace:

```
#!/usr/bin/env python

import MySQLdb

print "Content-Type: text/html\n"
print
print "<html><head><title>Libros</title></head>"
print "<body>"
print "<h1>Libros</h1>"
print "<ul>"

connection = MySQLdb.connect(user='yo', passwd='dejamentrar', db='books.db')
cursor = connection.cursor()
cursor.execute("SELECT nombre FROM libros ORDER BY fecha DESC
   LIMIT 10")
for row in cursor.fetchall():
   print "<li>%s</li>" % row[0]

print "</ul>"
print "</body></html>"
connection.close()
```

Este código es sencillo de entender. Primero imprime una línea de "Content-Type", seguido de una línea en blanco, tal como requiere CGI. Imprime el HTML introductorio, se conecta a la base de datos y ejecuta una consulta que obtiene los diez últimos libros publicados, de una tabla llamada libros. Hace un bucle sobre esos libros y genera una lista HTML desordenada. Finalmente imprime el código para cerrar el HTML y cierra la conexión con la base de datos.

Con una página única y poco dinámica como esta, el enfoque desde cero no es necesariamente malo. Por un lado, este código es sencillo de comprender – incluso un desarrollador novato puede leer estas líneas de Python y entender todo lo que hace el script, de principio a fin. No hay más nada que aprender; no hay más código para leer. También es sencillo de utilizar: sólo guarda este código en un archivo llamado ultimoslibros.cgi, sube ese archivo a un servidor Web y visita esa página con un navegador.

Sin embargo a medida que una aplicación Web crece más allá de lo trivial, este enfoque se desmorona y te enfrentas a una serie de problemas:

- ¿Qué sucede cuando múltiples páginas necesitan conectarse a la base de datos? Seguro que ese código de conexión a la base de datos no debería estar duplicado en cada uno de los scripts CGI, así que la forma pragmática de hacerlo sería refactorizarlo en una función compartida.

- ¿Debería un desarrollador realmente tener que preocuparse por imprimir la línea de "Content-Type" y acordarse de cerrar la conexión con la base de datos? Este tipo de código repetitivo reduce la productividad del programador e introduce la oportunidad para que se cometan errores. Estas tareas de configuración y cierre estarían mejor manejadas por una infraestructura común.

- ¿Qué sucede cuando este código es reutilizado en múltiples entornos, cada uno con una base de datos y contraseñas diferentes? En ese punto, se vuelve esencial alguna configuración específica del entorno.

- ¿Qué sucede cuando un diseñador Web que no tiene experiencia programando en Python desea rediseñar la página? Lo ideal sería que la lógica de la página – la búsqueda de libros en la base de datos – esté separada del código HTML de la página, de modo que el diseñador pueda hacer modificaciones sin afectar la búsqueda.

Precisamente estos son los problemas que un framework Web intenta resolver. Un framework Web provee una infraestructura de programación para tus aplicaciones, para que puedas concentrarte en escribir código limpio y de fácil mantenimiento sin tener que reinventar la rueda. En resumidas cuentas, eso es lo que hace Django.

El patrón de diseño MVC

Comencemos con un ejemplo rápido, que demuestra la diferencia entre el enfoque anterior y el empleado al usar un framework Web. Así es como se podría escribir el código CGI anterior usando Django:

models.py

```
from django.db import models
'''Las tablas de la base de datos'''

class Libro(models.Model):
    nombre = models.CharField(max_length=50)
    fecha = models.DateField()
```

views.py

```
from django.shortcuts import render_to_response
from models import Libro

def ultimos_libros(request):

    '''La parte lógica'''
    lista_libros = Libro.objects.order_by('-fecha')[:10]
    return render_to_response('ultimos-libros.html',   {'lista_libros': lista_libros})
```

urls.py

```
from django.conf.urls import url
import views

# La configuración de la URL
urlpatterns = [
    url(r'^ultimos_libros/$', views.ultimos_libros),
]
```

ultimos_libros.html

```
{# La plantilla #}
<html><head><title>Libros</title></head>
<body>
    <h1>Libros</h1>
<ul>
    {% for libro in lista_libros %}
        <li>{{ libro.nombre }}</li>
    {% endfor %}
</ul>
</body></html>
```

No es todavía necesario preocuparse por los detalles sobre cómo funciona esto – tan sólo queremos que te acostumbres al diseño general. Lo que hay que notar principalmente en este ejemplo, son las *cuestiones de separación*:

- El archivo **models.py** contiene una descripción de la tabla de la base de datos, como una clase Python. A esto se lo llama el *modelo*. Usando esta clase se pueden crear, buscar, actualizar y borrar entradas de tu base de datos usando solo código Python en lugar de escribir declaraciones SQL repetitivas.

- El archivo **views.py** contiene la lógica de la página, en la función ultimos_libros() . A esta función se la denomina *vista*.

- El archivo **urls.py** específica qué vista es llamada según el patrón URL. En este caso, la URL /ultimos_libros/ será manejada por la función ultimos_libros(). En otras palabras, si el nombre de nuestro dominio es example.com, cualquier visita a la URL http://example.com/ultimos_libros/ llamara a la función ultimos_libros().

- El archivo **ultimos_libros.html** es una plantilla HTML especial, que describe el diseño de la página. Usa el lenguaje de plantillas de Django, con declaraciones básicas y lógicas por ejemplo: {% for libro in lista_libros %}.

Tomadas en su conjunto, estas piezas se aproximan al patrón de diseño Modelo-Vista-Controlador (MVC). Dicho de manera más fácil, MVC define una forma de desarrollar software en la que el código para definir y acceder a los datos (el modelo) está separado del pedido lógico de asignación de ruta (el controlador), que a su vez está separado de la interfaz del usuario (la vista).

Una ventaja clave de este enfoque es que los componentes tienen un *acoplamiento débil* entre sí. Eso significa que cada pieza de la aplicación Web que funciona sobre Django tiene un único propósito clave, que puede ser modificado independientemente sin afectar las otras piezas. Por ejemplo, un desarrollador puede cambiar la URL de cierta parte de la aplicación sin afectar la implementación subyacente. Un diseñador puede cambiar el HTML de una página sin tener que tocar el código Python que la renderiza. Un administrador de base de datos puede renombrar una tabla de la base de datos y especificar el cambio en un único lugar, en lugar de tener que buscar y reemplazar en varios archivos.

En este libro, cada componente tiene su propio capítulo. Por ejemplo, el c*apítulo3* trata sobre las vistas, el c*apítulo4* sobre las plantillas y el c*apítulo5* sobre los modelos.

Historia de Django

Antes de empezar a escribir código, deberíamos tomarnos un momento para explicar la historia de Django. Y para mostrar cómo se hacen las cosas *sin* usar atajos, esto nos ayudara a entenderlos mejor. Es útil entender *por qué* se creó el framework, ya que el conocimiento de la historia pone en contexto la razón por la cual Django trabaja de la forma en que lo hace.

Si has estado creando aplicaciones Web por un tiempo, probablemente estés familiarizado con los problemas del ejemplo CGI presentado con anterioridad. El camino clásico de un desarrollador Web es algo como esto:

1. Escribir una aplicación Web desde cero.
2. Escribir otra aplicación Web desde cero.
3. Darse cuenta de que la aplicación del paso, 1 tiene muchas cosas en común con la aplicación del paso 2.

4. Refactorizar el código para que la aplicación 1, comparta código con la aplicación 2.
5. Repetir los pasos 2-4 varias veces.
6. Darse cuenta de que acabamos de inventar un framework.

Así es precisamente como surgió Django.

Django nació naturalmente de aplicaciones de la vida real escritas por un equipo de desarrolladores Web en Lawrence, Kansas. Nació en el otoño boreal de 2003, cuando los programadores Web del diario *Lawrence Journal-World*, Adrian Holovaty y Simon Willison, comenzaron a usar Python para crear sus aplicaciones.

El equipo de The World Online, responsable de la producción y mantenimiento de varios sitios locales de noticias, prosperaban en un entorno de desarrollo dictado por las fechas límite del periodismo. Para los sitios –incluidos LJWorld.com, Lawrence.com y KUsports.com los periodistas (y los directivos) exigían que se agregaran nuevas características y que aplicaciones enteras se crearan a una velocidad vertiginosa, a menudo con sólo días u horas de preaviso. Es así que Adrian y Simon desarrollaron por necesidad un framework de desarrollo Web que les ahorrara tiempo – era la única forma en que podían crear aplicaciones mantenibles en tan poco tiempo.

En el verano de 2005, luego de haber desarrollado este framework hasta el punto en que estaba haciendo funcionar la mayoría de los sitios de World Online, el equipo de World Online, que ahora incluía a Jacob Kaplan-Moss, decidió liberar el framework como software de código abierto. Lo liberaron en julio de 2005 y lo llamaron Django, por el guitarrista de jazz "Django Reinhardt".

Hoy en día, Django es un proyecto estable y maduro, de código abierto con cientos de miles de colaboradores y usuarios de todo el mundo. Dos de los desarrolladores originales de Worl Online ("Los benevolentes dictadores vitalicios" Adrian y Jacob) siguen aportando una guía centralizada para el crecimiento del framework, por lo que es más un equipo de colaboración comunitario.

Esta historia es relevante porque ayuda a explicar dos cuestiones clave. La primera es el "punto dulce" de Django. Debido a que Django nació en un entorno de noticias, ofrece varias características (en particular la interfaz administrativa, tratada en el capítulo 6, que son particularmente apropiadas para sitios de "contenido" – sitios como eBay, craigslist.org y washingtonpost.com que ofrecen información basada en bases de datos. (De todas formas, no dejes que eso te quite las ganas a pesar de que Django es particularmente bueno para desarrollar esa clase de sitios, eso no significa que no sea una herramienta efectiva para crear cualquier tipo de sitio Web dinámico. Existe una gran diferencia entre ser *particularmente efectivo* para algo y *no ser particularmente efectivo* para otras cosas).

La segunda cuestión a resaltar es cómo los orígenes de Django le han dado forma a la cultura de su comunidad de código abierto. Debido a que Django fue extraído de código de la vida real, en lugar de ser un ejercicio académico o un producto comercial, está especialmente enfocado en resolver problemas de desarrollo Web con los que los desarrolladores de Django se han encontrado – y con los que continúan encontrándose. Como resultado de eso, Django es continuamente mejorado. Los desarrolladores del framework tienen un alto grado de interés en asegurarse de que Django les ahorre tiempo a los desarrolladores, produzca aplicaciones que sean fáciles de mantener y rindan bajo mucha carga. Aunque existen otras razones, los desarrolladores están motivados por sus propios deseos egoístas de ahorrarse tiempo a ellos mismos y disfrutar de sus trabajos.

Como leer este libro

Al escribir este libro, tratamos de alcanzar un balance entre legibilidad y referencia, con una tendencia a la legibilidad. Nuestro objetivo, como se mencionó anteriormente, es hacerte un experto en Django, y creemos que la mejor manera de enseñar es a través de la prosa y numerosos ejemplos, en vez de proveer un exhaustivo pero inútil catálogo de las características de Django (Como alguien dijo una vez, no puedes esperar enseñarle a alguien cómo hablar simplemente enseñándole el alfabeto).

Con esto en mente, te recomendamos que leas los capítulos del 1 al 7 en orden. Ellos forman los fundamentos básicos sobre la forma en que se usa Django; una vez que los hayas leído, serás capaz de construir sitios Web que funcionen sobre Django. Los capítulos 7 al 12, muestran característica avanzadas del framework, los capítulos restantes, están enfocados en características específicas de Django y pueden ser leídos en cualquier orden.
Los apéndices son para referencia. Que junto a la documentación libre disponible en http://www.djangoproject.com/, son probablemente los documentos que tendrás que leer de vez en cuando, para recordar la sintaxis o buscar un resumen rápido de lo que hace ciertas partes de Django, no explicadas aquí.

Conocimientos Requeridos

Los lectores de este libro deben comprender las bases de la programación orientada a objetos e imperativa: estructuras de control (por ejemplo: if, while, for)), estructuras de datos (listas, hashes/diccionarios), variables, clases y objetos.

La experiencia en desarrollo Web es, como podrás esperar, muy útil, pero no es requisito para leer este libro. A lo largo del mismo, tratamos de promover las mejores prácticas en desarrollo Web para los lectores a los que les falta este tipo de experiencia.

Conocimientos de Python requeridos

En esencia, Django es sencillamente una colección de bibliotecas escritas en el lenguaje de programación Python. Para desarrollar un sitio usando Django escribes código Python que utiliza esas bibliotecas. Aprender Django, entonces, es sólo cuestión de aprender a programar en Python y comprender cómo funcionan las bibliotecas de Django.

Si tienes experiencia programando en Python, no deberías tener problema en meterte de lleno. En conjunto, el código Django no produce "magia negra" (es decir, trucos de programación cuya implementación es difícil de explicar o entender). Para ti, aprender Django será sólo cuestión de aprender las convenciones y APIs de Django.

Si no tienes experiencia programando en Python, te espera una grata sorpresa. Es fácil de aprender y muy divertido de usar. A pesar de que este libro no incluye un tutorial completo de Python, sí hace hincapié en las características y funcionalidades de Python cuando se considera apropiado, particularmente cuando el código no cobra sentido de inmediato. Aún así, recomendamos leer el tutorial oficial de Python, disponible en http://pyspanishdoc.sourceforge.net/tut/tut.html o su versión más reciente en inglés en http://docs.python.org/tut/. También recomendamos el libro libre y gratuito de Mark Pilgrim *Inmersión en Python*, disponible en http://es.diveintopython.org/ y publicado en inglés en papel por Apress.

Versión requerida de Django

Este libro cubre la versión 1.8 y superior.

El equipo de desarrolladores de Django, trata en la medida de lo posible de mantener compatibilidad con versiones anteriores, sin embargo ocasionalmente, se introducen algunos cambios drásticos e incompatibles con versiones anteriores. En cada lanzamiento estos cambios son cubiertos en las notas del lanzamiento, que se pueden encontrar en: https://docs.djangoproject.com/en/dev/releases/2.X

Nuevas características de Django

Tal como mencionamos anteriormente, Django es mejorado con frecuencia, y probablemente tendrá un gran número de nuevas –e incluso esenciales características para cuando este libro sea publicado. Por ese motivo, nuestro objetivo como autores de este libro es doble:

- Asegurarnos que este libro sea "a prueba de tiempo" tanto como nos sea posible, para que cualquier cosa que leas aquí todavía sea relevante en futuras versiones de Django.
- Actualizar este libro continuamente en el sitio Web en inglés, http://www.djangobook.com/, para que puedas acceder a la mejor y más reciente documentación tan pronto como la escribamos.

Si quieres implementar con Django algo que no está explicado en este libro, revisa la versión más reciente de este libro en el sitio Web y también revisa la documentación oficial de Django, para obtener detalles más completos.

Obtener ayuda

Para obtener ayuda con cualquier aspecto de Django – desde instalación y diseño de aplicaciones, hasta diseño de bases de datos e implementaciones siéntete libre de hacer preguntas online.

En la lista de correo de usuarios de Django (en inglés) se juntan miles de usuarios para preguntar y responder dudas. Suscríbete gratuitamente en http://www.djangoproject.com/r/django-users.

El canal de IRC de Django donde los usuarios de Django se juntan a chatear y se ayudan unos a otros en tiempo real. Únete a la diversión en #django (inglés) o #django-es (español) en la red de IRC Freenode.

¿Qué sigue?

A continuación, en el capítulo 2 utilizaremos Django, explicaremos su instalación y la configuración inicial.

CAPÍTULO 2

Empezando

Instalar Django es un proceso que consta de varios pasos, debido a las múltiples partes móviles de las que constan los entornos modernos de desarrollo Web. En este capítulo se explican las situaciones más comunes de instalación del framework y algunas de sus dependencias.

Debido a que Django es "solo" código Python, se puede utilizar en cualquier sistema que corra Python, ¡incluyendo algunos teléfonos celulares! Por lo que este capítulo cubre los escenarios más comunes para su instalación. Asumiremos que quieres instalarlo en una computadora de escritorio/laptop o en un servidor.

Más adelante, en el c*apítulo12* te mostraremos, cómo desplegar Django en un sitio de producción.

Instalar Python

Django está escrito totalmente en código Python, por lo tanto lo primero que necesitas para usarlo, es asegurarte de que tienes instalada una versión apropiada de Python.

El núcleo del framework en la versión 1.4 trabaja con cualquier versión de Python superior a la 2.5 y 2.7. Django 1.5 y Django 1.6, requieren Python 2.6.5 como mínimo. A partir de este lanzamiento Python 3 es oficialmente soportado. Recomendamos ampliamente utilizar las últimas y menores versiónes de cada lanzamiento de Python soportados (2.7.X, 3.2.X, 3.3.X y 3.4.X). Toma en cuenta que Django 1.6 es la última versión que dará soporte a Python 2.6; ya que requiere como mínimo Python 2.7.

Si no estás seguro, sobre que versión de Python instalar y tienes completa libertad para decidir, busca una de las últimas versiónes de la serie 2, en especial la versión 3 o superior. Aunque Django trabaja bien con cualquiera de estas versiónes, las últimas versiónes de la serie 3 proveen mejores características.

Django y Python3

A partir de la versión 1.7, Django admite oficialmente Python 3, 3.2, 3.3, y 3.4. Por lo que si estas empezando un nuevo proyecto y las dependencias que piensas usar, trabajan bien en Python 3 deberías usar Python 3. De cualquier forma si no la haces, no deberías tener ningún problema con el funcionamiento de tu código.

A partir de Django 1.6, el soporte para Python 3 es considerado estable y puede ser usado de forma segura en producción. Es por ello que la comunidad está empezando a migrar paquetes de terceros y aplicaciones a Python 3, por lo que es una buena idea empezar a usarlo, ya que es más sencillo escribir código para Python 3 y luego hacerlo compatible con la serie 2, que viceversa.

Las nuevas versiones de Python, de la serie 3 son más rápidas y contienen más características que pueden ser muy útiles en tus proyectos Django, sin embargo si quieres usar una versión en especifico, toma en cuenta la siguiente tabla, para usar una versión de Python adecuada a la versión de Django que quieras usar:

Versión Django	Versión Python
1.4	2.5, 2.6, 2.7
1.5	2.6, 2.7 y 3.2, 3.3 (experimental)
1.6	2.6, 2.7 y 3.2, 3.3
1.7	2.7 y 3.2, 3.3, 3.4
1.8	2.7 y 3.2, 3.3, 3.4
1.9	2.7 y 3.2, 3.3, 3.4

Tabla 2.1 Compatibilidad entre Python y Django

Instalación

Si estás usando Linux o Mac OS X probablemente ya tienes instalada alguna versión de python o dos (Python 2.7 y Python3). Escribe python o python3 en una terminal. Si ves algo así, Python está instalado:

```
Python 3.4.0 (default, Apr 11 2014, 13:05:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Si ves un error como: “command not found” u “orden no encontrada”, necesitas primero bajar e instalar Python. Para empezar puedes encontrar instrucciones más detalladas en la página de descargas oficial de Python, disponible online en:
http://www.python.org/download/. La instalación es rápida y fácil.

Instalar Django

En cualquier momento, puedes disponer de dos versiones distintas de Django:

1. El lanzamiento oficial más reciente.
2. La versión de desarrollo.

La versión que decidas instalar dependerá de tus prioridades. Si quieres una versión estable, probada y lista para producción, instala la primera opción, sin embargo si quieres obtener las últimas y mejores características y si tal vez te gustaría contribuir con Django mismo, usa la segunda opción si no te importa mucho la estabilidad.

Nosotros recomendamos encarecidamente usar la versión oficial, pero siempre es importante conocer que existe una versión de desarrollo, ya que como se menciona en la documentación, esta está disponible para cualquier miembro de la comunidad de forma libre.

En esta sección explicaremos algunas opciones de instalación: instalar un lanzamiento oficial, ya sea manualmente o usando pip y explicaremos la forma de instalar la versión de desarrollo, desde el repositorio oficial Git.

Instalar un lanzamiento oficial

Los lanzamientos oficiales tienen un número de versión, tal como 1.7, 1.8 o 1.9, la última versión siempre está disponible en la página de descargas del proyecto en: http://www.djangoproject.com/download/.

Si estas usando alguna distribución de Linux, la mayoría incluye un paquete de Django, por lo que siempre es buena idea usar la versión que se distribuye para tu plataforma e instalar Django con el gestor de paquetes predeterminado. De esta forma la instalación no traerá problemas de seguridad al resto del sistema de paquetes, actualizando paquetes de forma innecesaria, además los gestores se encargan automáticamente de instalar las dependencias necesarias (como los conectores para la base de datos).

Si no tienes acceso a una versión pre-empaquetada, puedes descargar e instalar el framework manualmente. Para hacerlo primero descarga el tarball, que se llamará algo así como Django-version.tar.gz (No importa cuál sea el directorio local que elijas para la descarga, ya que el proceso de instalación se encargara de colocar Django en el lugar correcto). Descomprímelo con alguna utilidad como: tar xzvf Django-*.tar.gz, cámbiate al directorio recién creado con: cd Django-* y usa el comando: setup.py install o python setup.py install, tal y como instalarías cualquier otra librería Python (No olvides usar sudo o privilegios de administrador).

En sistemas tipo Unix, esta es la forma en que se ve el proceso:

1. tar xzvf Django-2.0.tar.gz
2. cd Django-*
3. sudo python setup.py install

En Windows, recomendamos usar 7-Zip (http://www.djangoproject.com/r/7zip/) para manejar archivos comprimidos de todo tipo, incluyendo .tar.gz. Una vez que has descomprimido el archivo, ejecuta un shell de comandos, con privilegios de administrador y ejecuta el siguiente comando desde el directorio que empieza con Django-:

```
python setup.py install
```

Si eres algo curioso, te darás cuenta que la instalación de Django, lo que hace es instalarse en un directorio llamado site-packages –Un directorio de paquetes, donde Python busca las librerías de terceros. Usualmente está ubicado en el directorio /usr/lib/python3/site-packages/

Instalar un lanzamiento oficial con Pip

Una forma muy sencilla y automática, para instalar paquetes en Python, es usando un instalador independiente llamado pip. Si tienes instalado pip, lo único que necesitas, es utilizar una versión actualizada. (Ya que en algunos casos la instalación de Django no trabaja, con versiones *muy* antiguas de pip), pip es la opción recomendada para instalar Django, usando virtualenv y virtualenvwrapper.

Pip es un instalador de paquetes Python, usado oficialmente para instalar paquetes desde Python Package Index (PyPI). Con pip puedes instalar Django desde PyPI, si no tienes instalado pip instalalo y luego instala Django.

- Abre una terminal de comandos y ejecuta el comando easy_install pip. Este comando instalara pip en tu sistema (La version 3.4 de Python, lo incluye como el instalador por defecto).
- Opcionalmente puedes utilizar virtualenv o virtualenvwrapper. Estas herramientas, proveen un entorno independiente y aislado para ejecutar código Python, con lo cual es mas practico instalar paquetes, sin alterar el sistema, ya que permite instalar paquetes, sin privilegios de administrador y sin riesgos de actualizar dependencias y con la ventaja de usar el interprete Python que desees. Por lo que de ti depende que quieras aprender a usarlo o no.

- Si estas usando Linux, Mac OS X o algún otro sabor de Unix, usa el siguiente comando en una terminal para instalar Django: sudo pip install django. Si estas usando Windows, inicia el shell de comandos con privilegios de administrador y ejecuta el comando: pip install django. Este comando instalara Django en el directorio de paquetes de tu instalación por defecto de Python.

Si estas usando virtualenv, no necesitas usar sudo o privilegios de administrador, ya que los paquetes se instalaran en un directorio independiente, perteneciente al entorno de paquetes que crea el mismo virtualenv.

Instalar la "Versión de Desarrollo"

Django usa Git (http://git-scm.com) para el control del código fuente. La última versión de desarrollo está disponible desde el repositorio oficial en Git (https://github.com/django/django). Si quieres trabajar sobre la versión de desarrollo, o si quieres contribuir con el código de Django en sí mismo, deberías instalar Django desde el repositorio alojado en Git.

Git es libre, es un sistema de control de versiones de código abierto, usado por el equipo de Django para administrar cambios en el código base. Puedes descargar e instalar manualmente Git de http://git-scm.com/download, sin embargo es más sencillo instalarlo con el manejador de paquetes de tu sistema operativo (si es tu caso). Puedes utilizar un cliente Git para hacerte con el código fuente más actual de Django y, en cualquier momento, actualizar tu copia local del código fuente, conocido como un checkout local, para obtener los últimos cambios y mejoras hechas por los desarrolladores de Django.

Cuando uses un versión de desarrollo, debes tener en mente que cualquier cosa se puede romper en cualquier momento, por lo que no hay garantías de nada Una vez dicho esto, también debemos decirte que algunos miembros del equipo de Django ejecutan sitios de producción sobre la versión de desarrollo con el incentivo de mantenerlo estable.

Para obtener la última versión de desarrollo, sigue los siguientes pasos:

1. Asegúrate de tener instalado Git. Puedes obtener el software de http://git-scm.com/, también puedes encontrar excelente documentación en http://git-scm.com/documentation.

2. Clona el repositorio usando el comando:

```
git clone https://github.com/django/django djmaster
```

3. Localiza el directorio site-packages de tu instalación Python. Usualmente esta en el directorio: /usr/lib/python3/site-packages. Si no tienes idea de su localización, usa la línea de comandos y tipea:

```
python -c 'import sys, pprint; pprint.pprint(sys.path)'
```

   El resultado de la salida, incluirá el directorio de site-packages

   - Si no tienes un directorio site-packages, crea un archivo con el nombre: djmaster.pth edítalo y agrega la ruta completa al directorio djmaster. Por ejemplo, tu archivo puede contener una línea como la siguiente:
     - /path/to/djmaster
   - La carpeta: djmaster/django/bin en la ruta de tu sistema. Es el directorio que incluye las utilidades administrativas, tales como django-admin.py.

     Si los archivo .pth son nuevos para ti, puedes aprender más de ellos en http://www.djangoproject.com/r/python/site-module/.

Luego de descargar el código fuentes desde Git y haber seguido los pasos anteriores, no necesitas ejecutar setup.py install ¡Acabas de hacer este trabajo a mano!

Debido a que el código de Django cambia a menudo corrigiendo bugs y agregando funcionalidades, probablemente quieras actualizarlo con frecuencia o alguna que otra vez. Para actualizar el código, solo ejecuta el comando: `git pull origin master` desde el directorio djmaster.

Cuando ejecutes este comando, Git contactara https://github.com/django/django y automáticamente determinará si el código ha cambiado y actualizará tu versión local del código con cualquier cambio que se haya hecho desde la última actualización. Es muy bueno.

Finalmente, si estas usando la versión de desarrollo, necesitas conocer la versión de Django que estas ejecutando. Conocer el número de versión es importante en caso de que alguna vez necesites ayuda de la comunidad o para enviar alguna mejora del framework. En estos casos, es necesario informar sobre la revisión, esta revisión es también conocida como “commit”. Para encontrar el commit actual, tipea “git log -1” dentro del directorio django y busca el identificador después del “commit”. Este número cambia cada vez que Django cambia, se corrige algún error, se agrega alguna característica, se mejora la documentación o se implementa cualquier otra cosa.

Probando la instalación

Para obtener un poco más de retroalimentación, después del proceso de instalación, tomémonos un momento para probar la instalación. Usando la línea de comandos, cámbiate a otro directorio (*no* al directorio que contiene el directorio django) e inicia el intérprete interactivo tipeando *python3* o *python* dependiendo de la versión que estés usando. Si el proceso de instalación fue exitoso, deberías poder importar el modulo django:

```
>>> import django
>>> django.VERSION
(1,8, 'final', 0)
```

Ejemplos en el intérprete interactivo

El intérprete interactivo de Python es un programa de línea de comandos que te permite escribir un programa Python de forma interactiva. Para iniciarlo sólo ejecuta el comando python o python3 en la línea de comandos

Durante todo este libro, mostraremos ejemplos de código Python como si estuviesen escritos en el intérprete interactivo. El triple signo de mayor que (>>>) es el prompt de Python. Si estas siguiendo los ejemplos interactivamente, no copies estos signos.

Toma en cuenta que en el intérprete interactivo, las declaraciones multilinea son completadas con tres puntos (...).

Por ejemplo:

```
>>>from __future__ import print_function
>>> print ("""Esta es una
... cadena de texto que abarca
... tres lineas.""")
Esta es una
cadena de texto que abarca
tres lineas.
>>> def mi_funcion(valor):
...         print (valor)
>>> mi_funcion('hola')
hola
```

Estos tres puntos adicionales, al inicio de la línea son insertados por el interprete Python – No son parte de la entrada de datos. Los hemos incluido aquí para ser fieles a la salida del intérprete. Si copias estos ejemplos, asegúrate de no incluir estos tres puntos.

Observa también como importamos la función print_function del paquete future, para compatibilidad con Python 3. Esta es una solución perfecta para proyectos en los cuales se requiere mantener una compatibilidad entre distintas versiones de Python, sin tener que ramificar el código entre versiones 2.x y 3.x en específico, de esta forma es posible mantener el código independiente de la versión de Python.

Configurar la base de datos

En este punto, podrías escribir una aplicación Web usando Django, por que el único prerrequisito de Django es una instalación funcionando de Python. Sin embargo, este libro se centra en una de las mejores funcionalidades de Django, el desarrollo de sitios Web con soporte para *base de datos*, para ello necesitarás instalar un servidor de base de datos de algún tipo, para almacenar tus datos.

Si sólo quieres comenzar a jugar con Django, salta a la sección titulada “Empezar un proyecto” – pero créenos, querrás instalar una base de datos finalmente. Todos los ejemplos de este libro asumen que tienes una base de datos configurada.

Hasta el momento de escribir esto, Django admite oficialmente estos cuatro motores de base de datos:

PostgreSQL	http://www.postgresql.org/
SQLite 3	http://www.sqlite.org/
MySQL	http://www.mysql.com/
Oracle	http://www.oracle.com/

Además de las bases de datos oficialmente soportadas, existen otras ofrecidas por terceros, que permiten utilizar otras bases de datos con Django como son: SAP SQL, Anywhere, IBM, DB2, Microsoft SQL Server, Firebird, ODBC, ADSDB.

En general, todos los motores que listamos aquí trabajan bien con Django (Una notable excepción, es el soporte opcional para GIS, el cual es más poderoso usando PostgreSQL, que usando otras bases de datos.) Si no estás atado a ningún sistema y tienes la libertad para cambiarte a cualquier base de datos, nosotros recomendamos PostgreSQL, el cual logra un balance fino entre el costo, características, velocidad y estabilidad.

Configurar la base de datos, es un proceso que toma dos pasos:

- Primero, necesitas instalar y configurar la base de datos en sí mismo. Este proceso va mas allá de los alcances de este libro, pero cada una de las cuatro bases de datos que mencionamos anteriormente posee una vasta documentación en su sitio Web (Si usas un servicio de hosting compartido, lo más probable es que la base de datos ya este configurada y lista para usarse.)

- Segundo, necesitas instalar ciertas librerías Python para la base de datos en específico que vayas a utilizar (drivers). Estas forman parte del código de terceros, que permiten a Python conectarse a la base de datos. Repasararemos mas los requisitos en las siguientes secciones.

SQLite merece especial atención como herramienta de desarrollo. Es un motor de base de datos extremadamente simple y no requiere ningún tipo de instalación y configuración del servidor. Es por lejos el más fácil de configurar si sólo quieres jugar con Django, y viene incluido en la biblioteca estándar de Python.

En Windows, obtener los drivers binarios para cualquier base de datos es a veces un proceso complicado. Ya que sólo estás iniciándote con Django, recomendamos usar Python 3 y el soporte incluido para SQLite.

La compilación de drivers puede ser estresante.

Usar Django con PostgreSQL

Si estás utilizando PostgreSQL, necesitarás el paquete psycopg2 disponible en http://www.djangoproject.com/r/python-pgsql/. Toma nota de la versión de Python que estás usando; necesitarás esta información para descargar la versión apropiada.

Si estás usando PostgresSQL en Windows, puedes encontrar los binarios precompilados de psycopg en http://www.djangoproject.com/r/python-pgsql/windows/.

Si estas usando Linux, checa el instalador o gestor de paquetes que ofrece tú sistema, busca algo llamado “python-psycopg”, “psycopg-python”, “python-postgresql” o algo similar.

Usar Django con SQLite 3

Si quieres usar SQLite, estas de suerte, ya que no necesitas instalar nada, porque Python ofrece soporte nativo para SQLite, además Django ofrece por omisión usar esta configuración, por lo que puedes saltarte esta sección.

Usar Django con MySQL

Django requiere MySQL 4.0 o superior; la versión 3.x no admite subconsultas anidadas, ni algunas otras sentencias SQL perfectamente estándar.

También necesitas instalar el paquete MySQLdb disponible en:
http://www.djangoproject.com/r/python-mysql/.

Si estas usando Linux, checa el instalador o gestor de paquetes que ofrece tú sistema y busca algo llamado "python-mysql", "python-mysqldb", "mysql-python" o algo similar.

Django con Oracle

Django trabaja con versiones servidor de Oracle 9i o más alto.
Si estas usando Oracle necesitas instalar la librería cx_Oracle, disponible en
http://cx-oracle.sourceforge.net/. Usa una versión superior a la 4.31, pero evita la versión 5.0 ya que existe un error en la versión del controlador. La versión 5.0.1 corrige ese error, de cualquier forma usa en lo posible una versión superior.

Usar Django sin una base de datos

Como mencionamos anteriormente, Django actualmente no requiere una base de datos. Si sólo quieres usar este como un servidor dinámico de páginas que no use una base de datos, está perfectamente bien.

Con esto dicho, ten en cuenta que algunas de las herramientas extras de Django requieren una base de datos, por lo tanto si eliges no usar una base de datos, perderás estas utilidades. (Señalaremos estas utilidades a lo largo del libro).

Comenzar un proyecto

Una vez que has instalado Python, Django y (opcionalmente) una base de datos (incluyendo los controladores), puedes empezar a dar tus primeros pasos en el desarrollo de aplicaciones, creando un *proyecto*.

Un proyecto es una colección de configuraciones para una instancia de Django, incluyendo configuración de base de datos, opciones específicas de Django y configuraciones específicas de aplicaciones.

Si esta es la primera vez que usas Django, tendrás que tener cuidado de algunas configuraciones iníciales.

Primero crea un nuevo directorio para empezar a trabajar, por ejemplo algo como /home/username/djcode/.

¿Dónde debería estar este directorio?

Si has trabajado con PHP, probablemente pondrías el código debajo de la carpeta raíz del servidor Web (en lugares como /var/www). Con Django, no tienes que hacer esto. No es una buena idea poner cualquier código Python en la carpeta raíz del servidor Web, porque al hacerlo se arriesga a que la gente sea capaz de ver el código en la Web. Esto no es bueno para la seguridad.

Pon tu código en algún directorio **fuera** de la carpeta raíz, cámbiate al directorio que acabas de crear y ejecuta el siguiente comando:

```
django-admin.py startproject misitio
```

Este comando creara un directorio llamado misitio en el directorio actual.

Nota: django-admin.py debería estar en la ruta de búsqueda de tu sistema o PATH, si instalaste Django con la utilidad setup.py.

Si estas usando la versión de desarrollo, puedes encontrar django-admin.py en djmaster/django/bin. Como vas a utilizar con frecuencia django-admin.py considera agregarlo a tu PATH. En Unix, puedes hacer un enlace simbólico a /usr/local/bin usando un comando como sudo ln -s /path/to/django/bin/django-admin.py /usr/local/bin/django-admin.py. En Windows, puedes actualizar tu variable de entorno PATH de forma grafica.

Si instalaste Django a través del gestor de paquetes de tu distribución, django-admin.py o en sistemas Windows, ahora tienes un ejecutable llamado **django-admín**, A si que solo debes ejecutar el comando (omitiendo el .py):

```
django-admin.py startproject misitio
```

Si obtienes un mensaje como “permiso denegado”, al usar el comando django-admin.py startproject, necesitas cambiarle los permisos al archivo, para hacerlo, navega al directorio donde se instalo django-admin.py, (e.g., cd /usr/local/bin) y ejecuta el comando chmod +x django-admin.py.

El comando startproject crea un directorio de trabajo que contiene varios archivos:

```
misitio/
    manage.py
    misitio/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Estos archivos son los siguientes:

- **misitio/:** El directorio de trabajo externo misitio/, es solo un contenedor, es decir una carpeta que contiene nuestro proyecto. Por lo que se le puede cambiar el nombre en cualquier momento sin afectar el proyecto en sí.

- **manage.py**: Una utilidad de línea de comandos que te permite interactuar con un proyecto Django de varias formas. Usa manage.py help para ver lo que

puede hacer. No deberías editar este archivo, ya que este es creado en el directorio convenientemente para manejar el proyecto.

- **misitio/misitio/:** El directorio interno misitio/ contiene el paquete Python para tu proyecto. El nombre de este paquete Python se usara para importar cualquier cosa dentro del. (Por ejemplo import misitio.settings).

- **__init__.py:** Un archivo requerido para que Python trate el directorio misitio como un paquete o como un grupo de módulos. Es un archivo vacio y generalmente no necesitaras agregarle nada.

- **settings.py:** Las opciones/configuraciones para nuestro proyecto Django. Dale un vistazo, para que te des una idea de los tipos de configuraciones disponibles y sus valores predefinidos.

- **urls.py:** Declaración de las URLs para este proyecto de Django. Piensa que es como una "tabla de contenidos" de tu sitio hecho con Django.

- **wsgi.py**: El punto de entrada WSGI para el servidor Web, encargado de servir nuestro proyecto. Para más detalles consulta el c*apítulo12*.

Todos estos pequeños archivos, constituyen un proyecto Django, que puede albergar múltiples aplicaciones.

Si estas usando SQLite como base de datos, no necesitaras crear nada de antemano - la base de datos se creará automáticamente cuando esta se necesite.

Observa que la variable *INSTALLED_APPS*, hacia el final del archivo settings.py, contiene el nombre de todas las aplicaciones Django que están activadas en esta instancia de Django. Las aplicaciones pueden ser empacadas y distribuidas para ser usadas por otros proyectos.

De forma predeterminada INSTALLED_*APPS* contiene todas las aplicaciones, que vienen por defecto con Django:

- django.contrib.admin – La interfaz administrativa.
- django.contrib.auth – El sistema de autentificación.
- django.contrib.contenttypes – Un framework para tipos de contenidos.
- django.contrib.sessions – Un framework. para manejar sesiones
- django.contrib.messages – Un framework para manejar mensajes
- django.contrib.staticfiles – Un framework para manejar archivos estáticos.

Estas aplicaciones se incluyen por defecto, como conveniencia para los casos más comunes.

Algunas de estas aplicaciones hacen uso, de por lo menos una tabla de la base de datos, por lo que necesitas crear las tablas en la base de datos, antes de que puedas utilizarlas, para hacerlo entra al directorio que contiene tu proyecto (cd misitio) y ejecuta el comando siguiente, para activar el proyecto Django.:

```
python manage.py migrate
```

El comando migrate busca la variable *INSTALLED_APPS* y crea las tablas necesarias de cada una de las aplicaciones registradas en el archivo settings.py, que contiene todas las aplicaciones. Veras un mensaje por cada migración aplicada.

El servidor de desarrollo

Para obtener un poco de información y más retroalimentación, ejecuta el servidor de desarrollo de Django, para ver el proyecto en acción.

Django incluye un servidor Web ligero (Que es llamado con el comando "runserver") que puedes usar mientras estás desarrollando tu sitio. Incluimos este servidor para que puedas desarrollar tu sitio rápidamente, sin tener que lidiar con configuraciones de servidores Web para producción (por ejemplo, Apache) hasta que estés listo para producción.

Este servidor de desarrollo vigila tu código a la espera de cambios y se reinicia automáticamente, ayudándote a hacer algunos cambios rápidos en tu proyecto sin necesidad de reiniciar nada.

Para iniciar el servidor, entra en el directorio que contiene tu proyecto (cd misitio) si aún no lo has hecho y ejecuta el comando:

```
python manage.py runserver
```

Verás algo parecido a esto:

```
Performing system checks...

System check identified no issues (0 silenced).
February 16, 2014 - 21:17:11
Django version 1.8, using settings 'misitio.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

El comando runserver inicia el servidor de desarrollo en el puerto 8000, escuchando sólo conexiones locales. Ahora que el servidor está corriendo, visita la dirección http://127.0.0.1:8000/ con tu navegador Web. Verás una página de "Bienvenido a Django" sombreada con un azul pastel agradable. ¡Funciona! "Bienvenido a Django"

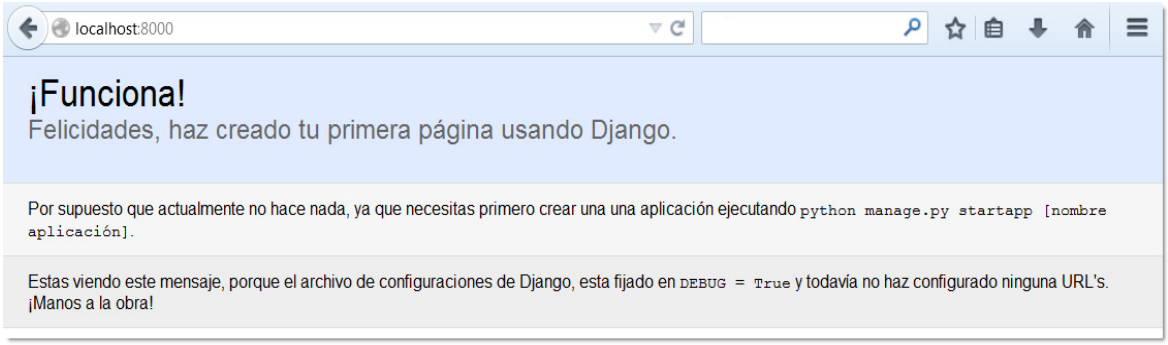


Imagen 2.1 ¡Página de bienvenida a django, ¡Funciona!

Aunque el servidor de desarrollo es extremadamente útil para, bueno; desarrollar, resiste la tentación de usar este servidor en cualquier entorno parecido a producción. El servidor de desarrollo puede manejar fiablemente una sola petición a la vez, y no ha pasado por una auditoría de seguridad de ningún tipo. Cuando sea el momento de

lanzar tu sitio, mira el *Capítulo12* para obtener más información, sobre cómo hacerlo con Django.

Cambiar el host y el puerto

Por defecto, el comando runserver inicia el servidor de desarrollo en el puerto 8000, escuchando sólo conexiones locales. Si quieres cambiar el puerto del servidor, pásaselo a este como un argumento en la línea de comandos:

```
python manage.py runserver 8080
```

También puedes cambiar las direcciones IP que escucha el servidor. Esto es utilizado especialmente si quieres compartir el desarrollo de un sitio con otros desarrolladores o miembros de tu equipo. Por ejemplo la dirección IP 192.148.1.103 hará que Django escuche sobre cualquier interfaz de red, permitiendo que los demás miembros del equipo puedan conectarse al servidor de desarrollo.

```
python manage.py runserver 192.148.1.103:8000
```

Cuando hagas esto, otras computadoras de tu red local, podrán conectarse a tu sitio, visitando la dirección IP directamente en sus navegadores por ejemplo usando: http://192.148.1.103:8000/ . (Ten en cuenta que para poder acceder a tu red, es necesario primero determinar la dirección IP de tu red local, en sistemas Unix, puedes usar el comando "ifconfig" en la línea de comandos o terminal, en sistemas Windows puedes conseguir esta misma información usando el comando "ipconfig")

¿Qué sigue?

Ahora que tienes todo instalado y el servidor de desarrollo funcionando, en el próximo capítulo *Los principios de las páginas Web dinámicas* escribirás algo de código, que muestra cómo servir páginas Web usando Django.

CAPÍTULO 3

Los principios básicos de las páginas Web dinámicas

En el capítulo anterior, explicamos cómo crear un proyecto en Django y cómo poner en marcha el servidor de desarrollo. Por supuesto, el sitio no hace nada útil todavía – sólo muestra el mensaje “It worked!”. Cambiemos eso. Este capítulo presenta cómo crear páginas web dinámicas con Django.

Tu primera pagina creada con Django

Como primer objetivo, vamos a crear una página web, que muestre por salida el famoso y clásico mensaje: “Hola mundo”.

Si quisiéramos publicar un simple “Hola mundo” en una página web, sin usar un framework, simplemente escribiríamos “Hola mundo” en un archivo de texto y lo llamaríamos "Hola_mundo", después lo subiríamos a alguna parte de nuestro servidor Web. Fíjate en el proceso, hemos especificado dos piezas de información acerca de la página web: tenemos el contenido que es la cadena "Hola mundo" y la URL que puede ser http://www.example.com/hola.html o tal vez http://www.example.com/archivos/hola.html si lo pusimos en un subdirectorio.

Con Django, es necesario especificar esas mismas cosas, pero de diferente modo. El contenido de la página será producido por la *función vista* y la URL se especificara en la *URLconf*.

Primero escribamos la vista “Hola mundo”, podríamos crear una aplicación Django para este propósito, pero lo haremos de forma manual, para conocer paso a paso el proceso de creación, mas adelante te mostraremos como crear aplicaciones de forma automática.

Tu primera vista creada con Django

Dentro del directorio misitio, el cual creamos en el capítulo anterior con el comando django-admin.py startproject, crea un archivo vacio llamado *views*.py en el mismo nivel que settings.py. Este modulo Python contendrá la vista que usaremos en este capítulo. Observa que no hay nada especial acerca del nombre views.py – A Django no le interesa como lo llames. Le dimos este nombre solo por convención y para beneficio de otros desarrolladores que lean nuestro código.

Nuestra vista “Hola mundo” será bastante simple. Esta es la función completa, la cual incluye las declaraciones que debemos escribir en un archivo llamado views.py:

views.py

```
from django.http import HttpResponse

def hola(request):
    return HttpResponse("Hola Mundo")
```

Repasemos el código anterior línea a línea:

- Primero, importamos la clase HttpResponse, la cual pertenece al módulo django.http. Necesitamos importar esta clase porque será usada posteriormente en nuestro código.

- Después, definimos una función llamada hola, la función vista.

- Cada función de vista o vista, toma al menos un parámetro llamado por convención request. El cual es un objeto que contiene información sobre la vista que llama a la página actual, la cual es una instancia de la clase django.http.HttpRequest. En este ejemplo, no hace nada el método request, no obstante siempre debe ser el primer parámetro de cualquier vista.

- Observa también que el nombre de la función no importa; ya que no tiene que ser nombrada de una determinada forma para que Django la reconozca. La llamamos hola, porque claramente indica lo que hace esta función, pero se podría haber llamado, hola_maravilloso_y_bello_mundo, o algo igualmente provocador. En la siguiente sección “Tu primera URLconf”, te mostraremos como le decimos a Django, que encuentre esta función.

La función que hemos creado, es una simple línea: que retorna un objeto HttpResponse que ha sido instanciado con el texto "Hola mundo", pero por ejemplo si quisiéramos mostrar por salida HTML directamente, lo podríamos hacer así, remplaza la vista anterior con esta:

views.py

```
from django.http import HttpResponse

HTML = """
<!DOCTYPE html>
<html lang="es">
<head>
<meta http-equiv="content-type" content="text/html;  charset=utf-8">
<meta name="robots" content="NONE,NOARCHIVE">
<title>Hola mundo</title>
<style type="text/css">
html * { padding:0; margin:0; }
body * { padding:10px 20px; }
body * * { padding:0; }
body { font:small sans-serif; }
body>div { border-bottom:1px solid #ddd; }
h1 { font-weight:normal; }
#summary { background: #e0ebff; }
</style>
</head>
<body>
<div id="summary">
<h1>¡Hola Mundo!</h1>
</div>
</body></html> """

def hola(request):
    return HttpResponse(HTML)
```

La lección principal que debes aprender aquí, es que una vista es solo una función Python, que toma como primer argumento una petición HttpRequest y retorna como respuesta una instancia de HttpResponse. Por lo que **una función en Python es una vista en Django**. (Hay excepciones, pero las veremos más adelante)

Tu primera URLconf creada con Django

En este punto, puedes ejecutar otra vez python manage.py runserver y veras de nuevo el mensaje "Bienvenido a Django", sin rastros de la vista "Hola mundo" que creamos anteriormente. Esto se debe a que nuestro proyecto misitio, no sabe nada acerca de esta vista, por lo que necesitamos decirle a Django explícitamente, como activar esta vista para una determinada URL (Continuando con la analogía que mencionamos anteriormente, sobre publicar archivos estáticos, esto sería como crear un archivo HTML, sin subirlo al directorio del servidor). Para enganchar, enlazar o apuntar a una determinada URL una función vista, usamos una URLconf.

Una **URLconf** es como una tabla de contenidos para tu sitio web hecho con Django. Básicamente, es un mapeo entre los patrones URL y las funciones de vista que deben ser llamadas por esos patrones URL. Es como decirle a Django, "Para esta URL, llama a este código, y para esta otra URL, llama a este otro código". Por ejemplo, "Cuando alguien visita la URL /hola/, llama a la función vista_hola() la cual está en el modulo Python views.py."

Cuando ejecutaste django-admin.py startproject en el capítulo anterior, el script creó automáticamente una URLconf por ti: el archivo urls.py. Por omisión, se verá así:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    # Ejemplos:
    # url(r'^$', 'misitio.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),
    url(r'^admin/', include(admin.site.urls)),
]
```

Repasemos el código anterior línea a línea:

- La primera línea importa las funciones: url e include, del modulo django.conf.urls, la función url es una tupla, donde el primer elemento es una expresión regular simple y el segundo elemento es la función de vista que se usa para ese patrón, mientras que la función include se encarga de importar módulos que contienen otras URLconf, al camino de búsqueda de Python, como una forma de "incluir" urls que pertenecen a otro paquete, en este caso al sitio administrativo, que viene activado por defecto (Esto lo veremos más adelante).

- Después tenemos a la función urlpatterns(), una variable que recibe los argumentos de las url en forma de lista, inclusive cadenas de caracteres vacías.

- Por defecto, todo lo que está en la URLconf está comentado e incluye algunos ejemplos de configuraciones comúnmente usados, a excepción del sitio

administrativo, el cual esta activado por omisión. (Para desactivarlo, solo es necesario comentarlo.)

- Si ignoramos el código comentado y las referencias a la interfaz administrativa, esto es esencialmente una URLconf:

```
from django.conf.urls import url

urlpatterns = [

]
```

El principal punto que debes notar aquí es la variable urlpatterns, la cual Django espera encontrar en tu módulo ROOT_URLCONF. Esta variable define el mapeo entre las URLs y el código que manejan esas URLs. Por defecto, todo lo que está en la URLconf está comentado – tu aplicación de Django es una pizarra blanca. (Como nota adicional, esta es la forma en la que Django sabía que debía mostrar la página "It worked!" en el capítulo anterior. Si la URLconf está vacía, Django asume que acabas de crear el proyecto, por lo tanto, muestra ese mensaje).

Para agregar una URL y una vista a la URLconf, solamente agrega un mapeo, es decir un enlace entre el patrón URL y la función vista a usar. Esta es la forma en la que enganchamos la vista hola a la URL:

urls.py

```
from django.conf.urls import url
from misitio.views import hola

urlpatterns = [
    url(r'^hola/$', hola),
]
```

(Nota que borramos todo el código que hace referencia a la interfaz administrativa, y dejamos la URLconf vacía, para mostrar cómo funcionan las vistas en Django, activaremos la interfaz administrativa en capítulos posteriores.)

Observa que hicimos dos cambios:

- Primero, importamos la vista hola, desde el modulo misitio/views.py que en la sintaxis de import de Python se traduce a misitio.views. (La cual asume que el paquete misitio/views.py, está en la ruta de búsqueda de Python o python path).

- Luego, agregamos la línea url(r'^hola/$', hola), a urlpatterns Esta línea hace referencia a un *URLpattern* – Una tupla de Python en dónde el primer elemento es una expresión regular simple y el segundo elemento es la función de vista que usa para manejar ese patrón. La url() puede tomar argumentos opcionales, los cuales cubriremos más a fondo en él *Capítulo8*,

Un detalle importante que hemos introducido aquí, es el carácter r al comienzo de la expresión regular. Esto le dice a Python que es una "cadena en crudo" – lo que permite que las expresiones regulares sean escritas sin demasiadas sentencias de escape tal como cadenas '\n' , la cual es una cadena que indica una nueva línea. Cuando agregamos la r hicimos una cadena en crudo, la cual Python no tratar de escapar con r'\n' una cadena de dos caracteres, la diagonal y la "n" minúscula. Para

evitar colisiones entre las diagonales que usa Python y las encontradas en las expresiones regulares, es fuertemente recomendado usar cadenas en crudo, cada vez que necesites definir una expresión regular en Python. Todos los patrones en este libro usan cadenas en crudo.

En resumidas cuentas, le estamos diciendo a Django que cualquier petición a la URL /hola/ sea manejada por la función de vista: /hola/ (y no, no tienen que llamarse igual)

🗒 Tu ruta de python o python path

Python path es la lista de directorios en tu sistema en donde Python buscará cuando uses la sentencia import de Python.

Por ejemplo, supongamos que tu Python path tiene el valor ['', '/usr/lib/python3.4/site-packages', '/home/username/djcode/']. Si ejecutas el código Python from foo import bar, Python en primer lugar va a buscar el módulo llamado foo.py en el directorio actual. (La primera entrada en el Python path, una cadena de caracteres vacía, significa "el directorio actual.") Si ese archivo no existe, Python va a buscar el módulo en /usr/lib/python3.4/site-packages/foo.py. Si ese archivo no existe, entonces probará en /home/username/djcode/foo.py. Finalmente, si *ese* archivo no existe, Python lanzará una excepción ImportError

Si estás interesado en ver el valor de tu Python path, abre un intérprete interactivo de Python y escribe:

```
>>> from __future__ import print_function
>>> import sys
>>> print (sys.path)
```

De nuevo, importamos la función print_function() del paquete future para mantener compatibilidad entre Python 2 y 3.

Generalmente no tienes que preocuparte por asignarle valores al Python path – Python y Django se encargan automáticamente de hacer esas cosas por ti entre bastidores. (Si eres un poco curioso, establecer el Python path es una de las primeras tareas que hace el archivo manage.py).

Vale la pena discutir un poco más la sintaxis que usada en el patrón ''URLpattern'', ya que no es muy obvio el ejemplo, si esta es la primera vez que tropiezas con las expresiones regulares:

- La r en r'^hola/$' significa que '^hola/$' es una cadena de caracteres en crudo de Python. Esto permite que las expresiones regulares sean escritas sin demasiadas sentencias de escape.

- Puedes excluir la barra al comienzo de la expresión '^hola/$' para que coincida con /hola/. Django automáticamente agrega una barra antes de toda expresión. A primera vista esto parece raro, pero una URLconf puede ser incluida en otra URLconf, y el dejar la barra de lado simplifica mucho las cosas. Esto se retoma en el *capítulo 8*.

- El patrón incluye el *acento circunflejo* (^) y el *signo de dólar* ($) estos son caracteres de la expresión regular que tienen un significado especial. El acento circunflejo significa que "requiere que el patrón concuerde con el inicio de la cadena de caracteres", y el signo de dólar significa que "exige que el patrón concuerde con el fin de la cadena".

- Este concepto se explica mejor con un ejemplo. Si hubiéramos utilizado el patrón '^hola/' (sin el signo de dólar al final), entonces *cualquier* URL que comience con hola/ concordaría, así como /hola/foo y /hola/bar, no sólo /hola/. Del mismo modo, si dejamos de lado el carácter acento circunflejo inicial ('hola/$'), el patrón coincidiría con *cualquier* URL que termine con hola/, así como /foo/bar/time/. Por lo tanto, usamos tanto el acento circunflejo como el signo de dólar para asegurarnos que sólo la URL /hola/ coincida. Nada más y nada menos.

- La mayor parte de los patrones URL, empiezan con el acento circunflejo (^) y terminan con el signo de dólar ($), esto es bueno, ya que permite una mayor flexibilidad para realizar concordancias más complejas y exactas.

Expresiones regulares

Las *Expresiones Regulares* (o *regexes*) son la forma compacta de especificar patrones en un texto. Aunque las URLconfs de Django permiten el uso de regexes arbitrarias para tener un potente sistema de definición de URLs, probablemente en la práctica no utilices más que un par de patrones regex. Esta es una pequeña selección de patrones comunes:

Símbolo	Coincide con
. (punto)	Cualquier carácter
\d	Cualquier dígito
[A-Z]	Cualquier carácter, A-Z (mayúsculas)
[a-z]	Cualquier carácter, a-z (minúsculas)
[A-Za-z]	Cualquier carácter, a-z (no distingue entre mayúscula y minúscula)
+	Una o más ocurrencias de la expresión anterior (ejemplo, \d+ coincidirá con uno o más dígitos)
[^/]+	Cualquier carácter excepto la barra o diagonal.
?	Cero o una ocurrencia (ejemplo \d? coincidirá con cero o un digito
*	Cero o más ocurrencias de la expresión anterior (ejemplo, \d* coincidirá con cero o más dígitos)
{1,3}	Entre una y tres (inclusive) ocurrencias de la expresión anterior (ejemplo \d{1,3} coincidirá con uno, dos o tres dígitos)

Para más información acerca de las expresiones regulares, mira el módulo http://www.djangoproject.com/r/python/re-module/.

Quizás te preguntes qué pasa si alguien intenta acceder a /hola. (*sin* poner la segunda barra o diagonal). Porque no concuerda con el patrón que definimos, sin embargo por defecto cualquier petición a cualquier URL que *no* contenga una barra final y que no concuerde con un patrón, será redireccionado a la misma URL con la diagonal final, siempre y cuando la variable APPEND_SLASH tenga asignado el valor True. (APPEND_SLASH, significa "Agrega una diagonal al final". Consulta el *apéndice D,,* si quieres ahondar más en este tema).

Si eres el tipo de persona que le gusta que todas sus URL, contengan una barra al final (como lo prefieren muchos desarrolladores de Django), todo lo que necesitas es agregar la barra a cada patrón URL o asignar True a la variable APPEND_SLASH. Si prefieres que tus URLs *no* contengan la barra o si quieres decidir esto en cada URL, agrega False a la variable APPEND_SLASH y pon las barras en tus patrones URL respectivamente, como lo prefieras.

La otra cosa que debes observar acerca de las URLconf es que hemos pasado la función vista hola como un objeto, sin llamar a la función. Esto es una característica

de Python (y otros lenguajes dinámicos): las funciones son objetos de primera clase, lo cual significa que puedes pasarlas como cualquier otra variable. ¡Qué bueno! ¿no?

Para probar nuestros cambios en la URLconf, inicia el servidor de desarrollo de Django, como hiciste en el *capítulo 2*, ejecutando el comando python manage.py runserver (Si no lo tenías corriendo.) El servidor de desarrollo automáticamente detecta los cambios en tu código de Python y recarga de ser necesario, así no tienes que reiniciar el servidor al hacer cambios). El servidor está corriendo en la dirección http://127.0.0.1:8000/, entonces abre tu navegador web y ve a la página http://127.0.0.1:8000/hola/.

Deberías ver la salida de tu vista de Django, con el texto "Hola mundo", en un tono azul.

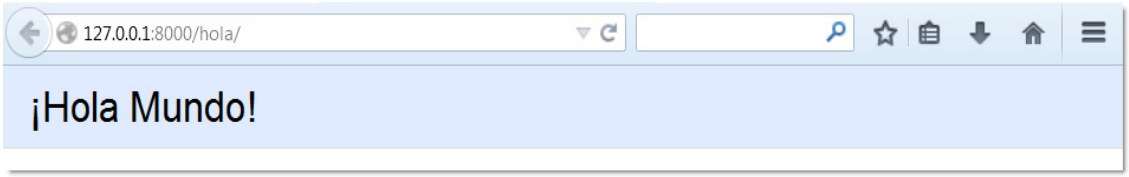


Imagen 3.1 Pagina "hola mundo", creada con Django.

¡Enhorabuena! Has creado tu primera página Web hecha con Django.

Algunas notas rápidas sobre errores 404

En las URLconf anteriores, hemos definido un solo patrón URL: el que maneja la petición para la URL /hola/. ¿Qué pasaría si se solicita una URL diferente?

Para averiguarlo, prueba ejecutando el servidor de desarrollo Django e intenta acceder a una página Web como http://127.0.0.1:8000/adios/ o http://127.0.0.1:8000/hola/directorio/ , o mejor como http://127.0.0.1:8000/ (la "raíz" del sitio). Deberías ver el mensaje "Page not found" ("Pagina no encontrada", ver la Figura siguiente). Es linda, ¿no? A la gente de Django seguro le gustan los colores pastel.

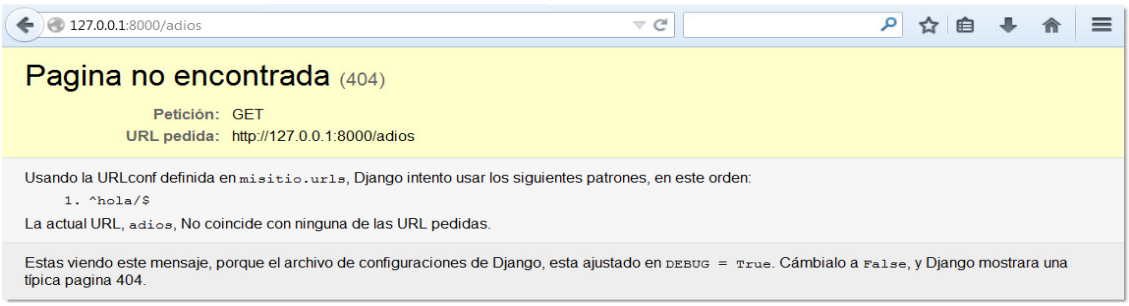


Imagen 3.2 Pagina de error 404

Django muestra este mensaje porque solicitaste una URL que no está definida en tu URLconf.

La utilidad de esta página va más allá del mensaje básico de error 404; nos dice también, qué URLconf utilizó Django y todos los patrones de esa URLconf. Con esa información, tendríamos que ser capaces de establecer porqué la URL solicitada lanzó un error 404.

Naturalmente, esta es información importante sólo destinada a ti, el administrador Web. Si esto fuera un sitio en producción alojado en Internet, no quisiéramos mostrar esta información al público. Por esta razón, la página "Page not found" es sólo mostrada si nuestro proyecto en Django está en modo de depuración (*debug mode*). Explicaremos cómo desactivar este modo más adelante. Por ahora, sólo diremos que todos los proyectos están en modo de depuración cuando los creamos, y si el proyecto no lo estuviese, se retornaría una respuesta diferente.

Algunas notas rápidas sobre la raíz del sitio

Como explicamos en la sección anterior, si estás viendo un mensaje de error 404 al acceder a la raíz de tu sitio http://127.0.0.1:8000/. Es porque Django no agrega mágicamente nada y las URLs no son un caso especial. Si quieres asignar un patrón a la raíz de tu sito, necesitas crear una vista y agregarla a la URL conf.

Cuando estés listo para implementar una vista para la raíz de tu sitio, usa el patrón '^$', el cual coincidirá con cualquier cadena vacía. Por ejemplo supongamos que creas una vista llamada 'raíz' la cual quieres usar como raíz de tu sitio:

```
from django.conf.urls import url
from misitio.views import raiz, hola

urlpatterns = [
    url(r'^$', raiz),
    url(r'^hola/$', hola),
    # ...
]
```

Cómo procesa una petición Django

Antes de crear una segunda vista, hagamos una pausa para aprender un poco más sobre la forma en Django trabaja. Especialmente analicemos cuando recibimos el mensaje "Hola mundo", al visitar la página http://127.0.0.1:8000/hola/ en el navegador web, esto es lo que Django hace tras bambalinas.

Todo comienza cuando el comando manage.py runserver importa un archivo llamado settings.py desde el directorio interno misitio. Este archivo contiene todo tipo de configuraciones opcionales para esta instancia de Django en particular, todas estas configuraciones están en mayúsculas: TEMPLATE_DIRS, DATABASES. Sin embargo una de las configuraciones más importantes es ROOT_URLCONF. La variable ROOT_URLCONF le dice a Django qué módulo de Python debería usar para la URLconf de este sitio Web.

¿Recuerdas cuando django-admin.py startproject creó el archivo settings.py y el archivo urls.py? Bueno, el archivo settings.py generado automáticamente contenía una variable ROOT_URLCONF que apunta al urls.py generado automáticamente. ¡Qué conveniente! Si abres el archivo settings.py; encontraras algo como esto:

```
ROOT_URLCONF = 'misitio.urls'
```

Este corresponde al archivo misitio/urls.py.

Cuando llega una petición –digamos, una petición a la URL /hola/ Django carga la URLconf apuntada por la variable ROOT_URLCONF. Luego comprueba cada uno de los patrones de URL, en la URLconf en orden, comparando la URL solicitada con un

patrón a la vez, hasta que encuentra uno que coincida. Cuando encuentra uno que coincide, llama a la función de vista asociada con ese patrón, pasando un objeto HttpRequest como primer parámetro de la función. (Veremos más de HttpRequest mas adelante).

Como vimos en el ejemplo anterior, la función de vista es responsable de retornar un objeto HttpResponse. Una vez que hace esto, Django hace el resto, convierte el objeto Python en una apropiada respuesta Web, que contiene las cabeceras HTTP y un cuerpo (es decir el contenido de la pagina Web.)

En resumen, el algoritmo sigue los siguientes pasos:

1. Se recibe una petición, por ejemplo a /hola/
2. Django determina la URLconf a usar, buscando la variable ROOT_URLCONF en el archivo de configuraciones.
3. Django busca todos los patrones en la URLconf buscando la primera coincidencia con /hola/.
4. Si encuentra uno que coincida, llama a la función vista asociada.
5. La función vista retorna una HttpResponse.
6. Django convierte el HttpResponse en una apropiada respuesta HTTP, la cual convierte en una página Web.

Ahora ya conoces lo básico sobre cómo hacer páginas Web con Django. Es muy sencillo, realmente – sólo tienes que escribir funciones de vista y relacionarlas con URLs mediante URLconfs. Podrías pensar que es lento enlazar las URL con funciones usando una serie de expresiones regulares, ¿pero te sorprenderás...!

Tu segunda Vista: Contenido dinámico

El ejemplo anterior, "Hola mundo" fue bastante instructivo y demostró la forma básica en la que trabaja Django, sin embargo no es un buen ejemplo de una página Web *dinámica* porque el contenido siempre es el mismo. Cada vez que visitemos /hola/, veremos la misma cosa; por lo que esta página, es más un archivo estático HTML.

Para nuestra segunda vista, crearemos algo más dinámico y divertido. Una página Web que muestre la fecha y la hora actual. Este es un buen ejemplo de una página *dinámica*, porque el contenido de la misma no es estático – ya que los contenidos cambian de acuerdo con el resultado de un cálculo (en este caso, el cálculo de la hora actual). Este segundo ejemplo no involucra una base de datos o necesita de entrada alguna, sólo muestra la salida del reloj interno del servidor. Es un poco más instructivo que el ejemplo anterior y demostrara algunos conceptos nuevos.

La vista necesita hacer dos cosas: calcular la hora actual y la fecha, para retornar una respuesta HttpResponse que contenga dichos valores. Si tienes un poco de experiencia usando Python, ya sabes que Python incluye un modulo llamado *datetime*, encargado de calcular fechas.

Esta es la forma en que se usa:

```
>>> from __future__ import print_function
>>> import datetime
>>> ahora = datetime.datetime.now()
>>> ahora
datetime.datetime(2014-10-16 17:36:30.493000)
>>> print (ahora)
```

```
2014-10-16 17:06:30.493000
```

El ejemplo es bastante simple y Django no necesita hacer nada. Ya que es solo código Python. (Es necesario hacer énfasis en que el código usado, "es solo Python" comparándolo específicamente con el código Django que usaremos. Para que no solo aprendas Django, sino no para que puedas aplicar tu conocimiento Python en otros proyectos, no necesariamente usando Django)

Para crear esta página, crearemos una *función de vista*, que muestre la fecha y la hora actual, por lo que necesitamos anclar la declaración datetime.datetime.now() dentro de la vista para que la retorne como una respuesta HttpResponse.

Esta es la vista que retorna la fecha y hora actual, como un documento HTML:

```
from django.http import HttpResponse
import datetime

def fecha_actual(request):
    ahora = datetime.datetime.now()
    html = <html><body><h1>Fecha:</h1><h3>%s<h/3></body></html>" % ahora
    return HttpResponse(html)
```

Así como la función hola, que creamos en la vista anterior, la función fecha_actual deve de colocarse en el mismo archivo views.py. Si estás siguiendo el libro y programando al mismo tiempo, notarás que el archivo views.py ahora contiene dos vistas. (Omitimos el HTML del ejemplo anterior sólo por claridad y brevedad). Poniéndolas juntas, veríamos algo similar a esto:

views.py

```
from django.http import HttpResponse
import datetime

def hola(request):
    return HttpResponse("Hola mundo")

def fecha_actual(request):
    ahora = datetime.datetime.now()
    html = "<html><body><h1>Fecha:</h1><h3>%s<h/3></body></html>" % ahora
    return HttpResponse(html)
```

Repasemos los cambios que hemos hecho al archivo views.py, para acomodar la función fecha_actual en la vista.

- Hemos agregado import datetime al inicio del modulo, el cual calcula fechas (Importamos el módulo datetime de la biblioteca estándar de Python) El módulo datetime contiene varias funciones y clases para trabajar con fechas y horas, incluyendo una función que retorna la hora actual.

- La nueva función fecha_actual calcula la hora y la fecha actual y almacena el resultado en la variable local ahora.

- La segunda línea de código dentro de la función construye la respuesta HTML usando el formato de cadena de caracteres de Python. El %s dentro de la cadena de caracteres es un marcador de posición, y el signo de porcentaje después de la cadena de caracteres, significa "Reemplaza el %s por el valor de la variable ahora." La variable ahora es técnicamente un objeto

datetime.datetime, no una cadena, pero %s el formato de cadenas de caracteres de Python lo convierte en algo así como: *"2014-10-16 17:36:30.493000"*. La cadena resultante será transformada en HTML de esta forma: *"<html><body>Hoy es 2014-10-16 17:36:30.493000. </body></html>"*. (Sí si si, el HTML es inválido, pero estamos tratando de mantener el ejemplo de forma simple y breve).

- Por último, la vista retorna un objeto HttpResponse que contiene la respuesta generada. Justo como en el ejemplo: Hola mundo.

Después de agregar la función a views.py, necesitamos agregar el patrón al archivo urls.py para decirle a Django que maneje esta vista. Algo como lo que hicimos con la función hola/ :

urls.py

```
from django.conf.urls import url
from misitio.views import hola, fecha_actual

urlpatterns = [
    url(r'^hola/$', hola),
    url(r'^fecha/$', fecha_actual),
]
```

Hemos hecho dos cambios aquí.

1. Primero, importamos la vista fecha_actual desde el módulo (misitio/views.py, que en la sintaxis de import de Python se traduce a misitio.views).
2. Segundo, y más importante agregamos un nuevo patrón que mapea la URL /fecha/ a la nueva función vista que hemos creado, agregando la línea url(r'^fecha/$', fecha_actual). Esta línea hace referencia a un *URLpattern* – una tupla de Python en dónde el primer elemento es una expresión regular simple y el segundo elemento es la función de vista que se usa para ese patrón.

Una vez que hemos escrito la vista y actualizado el patrón URL, ejecuta runserver y visita la página http://127.0.0.1:8000/fecha/ en tu navegador. Deberías poder ver la fecha y la hora actual.

ZONA HORARIA DE DJANGO

Dependiendo de tu computadora, de la fecha y la hora, la salida puede ser distinta. Esto se debe a que Django incluye una opción TIME_ZONE que por omisión es America/Chicago. Probablemente no es donde vivas, por lo que puedes cambiarlo en tu archivo de configuraciones settings.py. Puedes consultar http://en.wikipedia.org/wiki/List_of_tz_zones_by_name, para encontrar una lista completa de las zonas horario de todo el mundo.

URLconfs y el acoplamiento débil

Ahora es el momento de resaltar una parte clave de la filosofía detrás de las URLconf y detrás de Django en general: el principio de acoplamiento débil (*loose coupling*). Para explicarlo de forma simple: el acoplamiento débil es una manera de diseñar software aprovechando el valor de la importancia de que se puedan cambiar las piezas. Si dos piezas de código están débilmente acopladas (*loosely coupled*) los

cambios realizados sobre una de dichas piezas va a tener poco o ningún efecto sobre la otra.

Las URLconfs de Django son un claro ejemplo de este principio en la práctica. En una aplicación Web de Django, la definición de la URL y la función de vista que se llamará están débilmente acopladas; de esta manera, la decisión de cuál debe ser la URL para una función, y la implementación de la función misma, residen en dos lugares separados. Esto permite el desarrollo de una pieza sin afectar a la otra.

En contraste, otras plataformas de desarrollo Web acoplan la URL con el programa. En las típicas aplicaciones PHP (http://www.php.net/), por ejemplo, la URL de tu aplicación es designada por dónde colocas el código en el sistema de archivos. En versiones anteriores del framework Web Python CherryPy (http://www.cherrypy.org/) la URL de tu aplicación correspondía al nombre del método donde residía tu código. Esto puede parecer un atajo conveniente en el corto plazo, pero puede tornarse inmanejable a largo plazo.

Por ejemplo, consideremos la función de vista que escribimos antes, la cual nos mostraba la fecha y la hora actual. Si quieres cambiar la URL de tu aplicación – digamos, mover desde /fecha/ a /otrafecha/ – puedes hacer un rápido cambio en la URLconf, sin preocuparte acerca de la implementación subyacente de la función. Similarmente, si quieres cambiar la función de vista – alterando la lógica de alguna manera – puedes hacerlo sin afectar la URL a la que está asociada tu función de vista. Además, si quisiéramos exponer la funcionalidad de fecha actual en varias URL podríamos hacerlo editando el URLconf con cuidado, sin tener que tocar una sola línea de código de la vista así.

```
urlpatterns = [
    url(r'^hola/$', hola),
    url(r'^fecha/$', fecha_actual),
    url(r'^otrafecha/$', fecha_actual),
]
```

Este es el acoplamiento débil en acción. Continuaremos exponiendo ejemplos de esta importante filosofía de desarrollo a lo largo del libro.

Tu tercera vista: contenido dinámico

En la vista anterior fecha_actual, el contenido de la página – la fecha/hora actual – eran dinámicas, pero la URL (/fecha/) era estática. En la mayoría de las aplicaciones Web, sin embargo, la URL contiene parámetros que influyen en la salida de la página. Por ejemplo en una librería en línea, cada uno de los libros tendría una URL distinta así: /libro/243/ y /libro/81196/.

Siguiendo con los ejemplos anteriores, vamos a crear una tercera vista que nos muestre la fecha y hora actual con un adelanto de ciertas horas. El objetivo es montar un sitio en la que la página /fecha/mas/1/ muestre la fecha/hora, una hora más adelantada, la página /fecha/mas/2/ muestre la fecha/hora, dos horas más adelantada, la página /fecha/mas/3/ muestre la fecha/hora, tres horas más adelantada, y así sucesivamente.

A un novato se le ocurriría escribir una función de vista distinta para cada adelanto de horas, lo que resultaría en una URLconf como esta:

```
urlpatterns = [
    url(r'^fecha/$', fecha_actual),
    ulr(r'^fecha/mas/1/$', una_hora_adelante),
    url(r'^fecha/mas/2/$', dos_horas_adelante),
    url(r'^fecha/mas/3/$', tres_horas_adelante),
```

```
    url (r'^fecha/mas/4/$', cuatro_horas_adelante),
]
```

Claramente, esta línea de pensamiento es incorrecta. No sólo porque producirá redundancia entre las funciones de vista, sino también la aplicación estará limitada a admitir sólo el rango del horario definido – uno, dos, tres o cuatro horas. Si, de repente, quisiéramos crear una página que mostrara la hora cinco horas adelantada, tendríamos que crear una vista distinta y una línea URLconf, perpetuando la duplicación y la demencia. Aquí necesitamos algo de abstracción.

Algunas palabras acerca de las URLs bonitas

Si tienes experiencia en otra plataforma de diseño Web, como PHP o Java, es posible que estés pensado, "¡Oye, usemos un parámetro como una cadena de consulta!", algo así como /fecha/mas?horas=3, en el cual la hora será designada por el parámetro hora de la cadena de consulta de la URL (la parte a continuación de ?).

Con Django *puedes* hacer eso (pero te diremos cómo más adelante, si es que realmente quieres saberlo), pero una de las filosofías del núcleo de Django es que las URLs deben ser bonitas. La URL /fecha/mas/3 es mucho más limpia, más simple, más legible, más fácil de dictarse a alguien y ... Justamente más elegante que su homóloga forma de cadena de consulta.

Las URLs bonitas son un signo de calidad en las aplicaciones Web.

El sistema de URLconf que usa Django estimula a generar URLs agradables, haciendo más fácil el usarlas que él *no* usarlas.

Comodines en los patrones URL

Continuando con el diseño de nuestra aplicación, pongámosle un comodín al patrón URL, para que maneje las horas de forma arbitraria. Como ya se mencionó anteriormente, un patrón URL es una expresión regular; así que podemos usar una expresión regular \d+ como patrón, para que coincida con uno o más dígitos:

```
urlpatterns = [
    # ...
    url(r'^fecha/mas/\d+/$', horas_adelante),
    # ...
]
```

Este nuevo patrón coincidirá con cualquier URL que sea del tipo /fecha/mas/2/, /fecha/mas/25/, o también /fecha/mas/100000000000/. Bueno, ahora que lo pienso, podemos limitar el lapso máximo de horas a 99. Eso significa que queremos tener números de uno o dos dígitos en la sintaxis de las expresiones regulares, con lo que nos quedaría así \d{1,2}:

```
    url(r'^fecha/mas/\d{1,2}/$', horas_adelante),
```

(Hemos usado el carácter #... para comentar los patrones anteriores, solo por brevedad).

Nota: Cuando construimos aplicaciones Web, siempre es importante considerar el caso más descabellado posible de entrada, y decidir si la aplicación admitirá o no esa entrada. Aquí hemos limitado a los exagerados reconociendo lapsos de hasta 99 horas. Y, por cierto, *Los Limitadores exagerados*, aunque largo, sería un nombre fantástico para una banda musical.

Ahora designaremos el comodín para la URL, necesitamos una forma de pasar esa información a la función de vista, así podremos usar una sola función de vista para cualquier adelanto de hora. Lo haremos colocando paréntesis alrededor de los datos en el patrón URL que queramos guardar. En el caso del ejemplo, queremos guardar cualquier número que se anotará en la URL, entonces pongamos paréntesis alrededor de \d{1,2}:

```
    url(r'^fecha/mas/(\d{1,2})/$', horas_adelante),
```

Si estás familiarizado con las expresiones regulares, te sentirás como en casa aquí; estamos usando paréntesis para *capturar* los datos del texto que coincide.

La URLconf final, incluyendo las vistas anteriores, hola y fecha_actual, nos quedará así:

urls.py

```
from django.conf.urls import url
from misitio.views import hola, fecha_actual, horas_adelante

urlpatterns = [
    url(r'^hola/$', hola),
    url(r'^fecha/$', fecha_actual),
    url(r'^fecha/mas/(\d{1,2})/$', horas_adelante),
]
```

Ahora, vamos a escribir la función vista: horas_adelante. La vista horas_adelante es muy similar a la vista fecha_actual, que escribimos anteriormente, sólo que con una pequeña diferencia: tomará un argumento extra, el número de horas a mostrar por adelantado.

Agrega al archivo views.py lo siguiente:

```
from django.http import Http404, HttpResponse
import datetime

def horas_adelante(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt= datetime.datetime.now()+datetime.timedelta(hours=offset)
    html = "<html><body><h1>En %s hora(s), seran:</h1> <h3>
        %s</h3></body></html>" % (offset, dt)
    return HttpResponse(html)
```

Repasemos el código anterior línea a línea:

- Tal como hicimos en la vista fecha_actual, importamos la clase django.http.HttpResponse de Django y el módulo datetime de Python.

- La función de vista: horas_adelante, toma *dos* parámetros: request y offset.
  - **request** es un objeto HttpRequest, al igual que en hola y fecha_actual. Lo diremos nuevamente: cada vista *siempre* toma un objeto HttpRequest como primer parámetro.

  - **offset** es la cadena de caracteres capturada por los paréntesis en el patrón URL. Por ejemplo, si la petición URL fuera /fecha/mas/3/, entonces el offset debería ser la cadena de caracteres "3". Si la petición URL fuera /fecha/mas/21/, entonces el offset debería ser la cadena de caracteres "21". Nota que la cadena de caracteres capturada siempre es una cadena de caracteres, no un entero, incluso si se compone sólo de dígitos, como en el caso '21'.

- (Técnicamente, siempre debemos capturar *objetos unicode*, no bytestrings pero no te preocupes por esta distinción por el momento.)

- Decidimos llamar a la variable offset, pero puedes asignarle el nombre que quieras, siempre que sea un identificador válido para Python. El nombre de la variable no importa; todo lo que importa es lo que contiene el segundo parámetro de la función (luego de request). Es posible también usar una palabra clave, en lugar de posición, como argumentos en la URLconf. Eso lo veremos en detalle en el *capítulo 8*.

- Lo primero que hacemos en la función es llamar a int() sobre offset. Este método convierte el valor de una cadena de caracteres a entero.

- Toma en cuenta que Python lanzará una excepción ValueError si se llama a la función int() con un valor que no puede convertirse a un entero, como lo sería la cadena de caracteres "foo". En este ejemplo si nos topáramos con ValueError se lanzaría una excepción django.http.Http404, la cual cómo puedes imaginarte, da como resultado una **"Pagina no encontrada"** o un error 404.

  Algún lector atento se preguntara ¿Cómo podríamos levantar una excepción ValueError si estamos usando expresiones regulares en el patrón URL, ya que el patrón (\d{1,2}) captura solo dígitos y por consiguiente offset siempre será una cadena de caracteres conformada sólo por dígitos? La respuesta es que no debemos preocuparnos de atrapar la excepción, porque tenemos la certeza que la variable offset será una cadena de caracteres conformada sólo por dígitos. Esto ilustra otra ventaja de tener un URLconf: nos provee un primer nivel de validación de entrada. Por lo que es una buena práctica implementar funciones que implementen vistas que no hagan suposiciones sobre sus parámetros. ¿Recuerdas el acoplamiento débil?

- En la siguiente línea de la función, calculamos la fecha actual y la hora y le sumamos apropiadamente el número de horas. Ya habíamos visto el método datetime.datetime.now() de la vista fecha_actual el nuevo concepto es la forma en que se realizan las operaciones aritméticas sobre la fecha y la hora creando un objeto datetime.timedelta y agregándolo al objeto datetime.datetime. La salida se almacene en la variable dt.

- Esta línea muestra la razón por la que se llamó a la función int() con offset. En esta línea, calculamos la hora actual más las hora que tiene offset,

almacenando el resultado en la variable dt. La función datetime.timedelta requiere que el parámetro hours sea un entero.

- A continuación, construimos la salida HTML de esta función de vista, tal como lo hicimos en la vista anterior fecha_actual, con una pequeña diferencia en la misma linea, y es que usamos el formato de cadenas de Python con *dos* valores, no sólo uno. Por lo tanto, hay dos símbolos %s en la cadena de caracteres y la tupla de valores a insertar sería: (offset, dt).

- Finalmente, retornamos el HttpResponse del HTML – de nuevo, tal como hicimos en la vista fecha_actual.

Con esta función de vista y la URLconf escrita, ejecuta el servidor de desarrollo de Django (si no está corriendo), y visita http://127.0.0.1:8000/fecha/mas/5/, para verificar que lo que hicimos funciona. Luego prueba con http://127.0.0.1:8000/fecha/mas/15/.

Para terminar visita la pagina http://127.0.0.1:8000/fecha/mas/100/, para verificar que el patrón en la URLconf sólo acepta número de uno o dos dígitos, Django debería mostrar un error en este caso como "Page not found", tal como vimos anteriormente en la sección "Errores 404".

La URL http://127.0.0.1:8000/fecha/mas/ (*sin* horas designadas) debería también mostrar un error 404.

ORDEN PARA PROGRAMAR

En este ejemplo, primero escribimos el patrón URL y en segundo lugar la vista, pero en el ejemplo anterior, escribimos la vista primero y luego el patrón de URL. ¿Qué técnica es mejor? Bien, cada programador es diferente.

Si eres del tipo de programadores que piensan globalmente, puede que tenga más sentido que escribas todos los patrones de URL para la aplicación al mismo tiempo, al inicio del proyecto, y después el código de las funciones de vista. Esto tiene la ventaja de darnos una lista de objetivos clara, y es esencial definir los parámetros requeridos por las funciones de vista que necesitaremos desarrollar.

Si eres del tipo de programadores que les gusta ir de abajo hacia arriba, tal vez prefieras escribir las funciones de vista primero, y luego asociarlas a URLs. Esto también está bien.

Al final, todo se reduce a elegir qué técnica se amolda más a tu cerebro. Ambos enfoques son válidos.

Cómo procesa una petición Django: Detalles completos

Además del mapeo directo de URLs con funciones vista que acabamos de describir, Django nos provee un poco más de flexibilidad en el procesamiento de peticiones.

Acabamos de ver el flujo típico – resolución de una URLconf y una función de vista que devuelve un HttpResponse– sin embargo el flujo puede ser cortado o aumentado mediante middleware. Los secretos del middleware serán tratados en profundidad en él *capítulo15*, pero un esquema (ver Figura 3-3) te ayudará conceptualmente a poner todas las piezas juntas.

En resumen esto es lo que pasa:

- Cuando llega una petición HTTP desde el navegador, un *manejador* específico a cada servidor construye la HttpRequest, para pasarla a los componentes y manejar el flujo del procesamiento de la respuesta.

- El manejador luego llama a cualquier middleware de Petición o Vista disponible. Estos tipos de middleware son útiles para aumentar los objetos HttpRequest así como también para proveer un manejo especial a determinados tipos de peticiones. En el caso de que alguno de los mismos retornara un HttpResponse la vista no es invocada.

- Hasta a los mejores programadores se le escapan errores (*bugs*), pero el *middleware de excepción* ayuda a aplastarlos. Si una función de vista lanza una excepción, el control pasa al middleware de Excepción. Si este middleware no retorna un HttpResponse, la excepción se vuelve a lanzar.

- Sin embargo, no todo está perdido. Django incluye vistas por omisión para respuestas amigables a errores 404 y 500.

- Finalmente, el *middleware de respuesta* es bueno para el procesamiento posterior a un HttpResponse justo antes de que se envíe al navegador o haciendo una limpieza de recursos específicos a una petición.

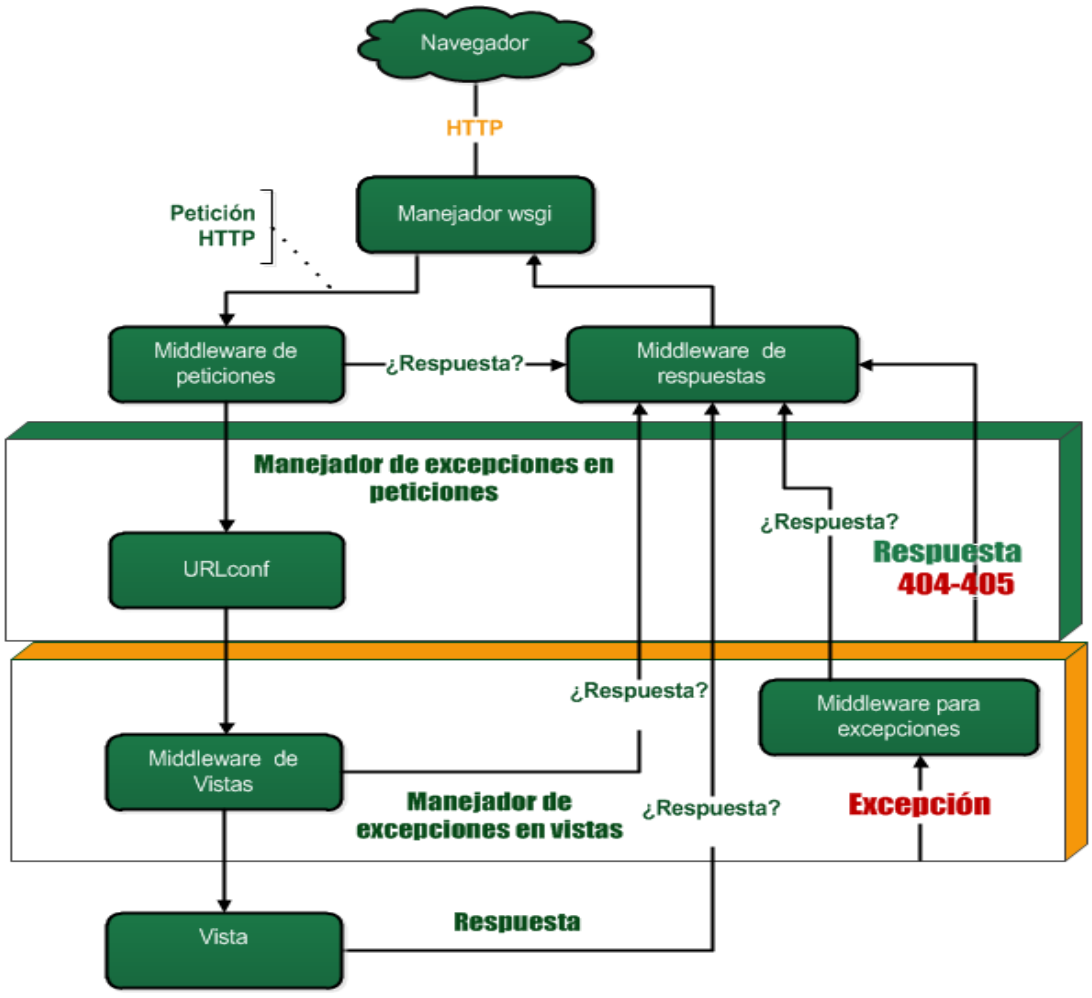


Imagen 3.3 El flujo completo de un petición y una respuesta en Django.

Páginas de error bonitas con Django

Tomémonos un momento para admirar la bonita aplicación web que hemos creado hasta ahora... y ahora ¡rompámosla! Introduzcamos deliberadamente un error de Python en el archivo views.py, comentando la línea offset = int(offset) de la vista horas_adelante:

```
from django.http import Http404, HttpResponse
import datetime

def horas_adelante(request, offset):
    #try:
    #   offset = int(offset)
    # except ValueError:
    #   raise Http404()
    dt= datetime.datetime.now()+datetime.timedelta(hours=offset)
    html = "<html><body><h1>En %s hora(s), seran:</h1>
        <h3> %s</h3></body></html>" % (offset, dt)
    return HttpResponse(html)
```

Ejecuta el servidor de desarrollo y navega a: http://127.0.0.1:8000/fecha/mas/3/. Verás una página de error con mucha información significativa, incluyendo el mensaje TypeError mostrado en la parte superior de la página: "unsupported type for timedelta hours component: unicode" o "string" en Python 3.

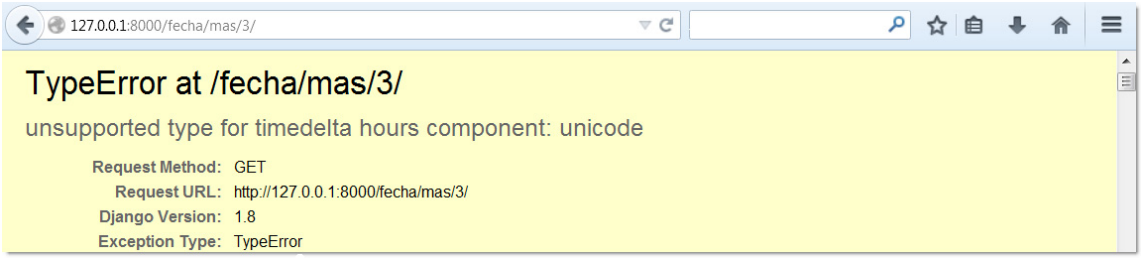


Imagen 3.4 Pagina de error bonita 404, mostrando información sobre el tipo de error.

¿Qué ha ocurrido? Bueno, la función datetime.timedelta espera que el parámetro hours sea un entero, y hemos comentado la línea de código que realiza la conversión del offset a entero.

Eso causa que datetime.timedelta lance un TypeError. Este es un típico error que todo programador comete en algún momento.

El punto específico de este ejemplo fue demostrar la página de error de Django.

Dediquemos un momento a explorar esta página y descubrir las distintas piezas de información que nos brinda:

- En la parte superior de la página se muestra la información clave de la excepción: el tipo y cualquier parámetro de la excepción (el mensaje "unsupported type" en este caso), y el archivo en el cuál la excepción fue lanzada, además de el número de línea que contiene el error.

- Abajo de la información clave de la excepción, la página muestra la traza de error o **traceback** de Python para dicha excepción. Esta es la traza estándar

que se obtiene en el interprete de Python, sólo que más interactiva y explicita. Por cada marco de pila, Django muestra el nombre del archivo, el nombre de la función/método, el número de línea y el código fuente de esa línea.

- Haz clic en la línea de código (en gris oscuro) para ver las líneas anteriores y posteriores a la línea errónea, lo que nos brinda un poco mas de contexto.

- Haz clic debajo de *"Locals vars"* (variables locles) sobre el marco de la pila para ver la tabla completa de todas las variables locales y sus valores, este marco muestra la posición exacta del código en el cual fue lanzada la excepción. Esta información de depuración es invaluable y muy privada.

- Observa que el texto *"Switch to copy-and-paste view"* (cambia a copiar y pegar) debajo de la cabecera de la traza de error. Haz clic en esas palabras, y la traza cambiará a una versión que te permitirá fácilmente copiar y pegar. Usa esto para cuando necesites compartir la traza de error de la excepción con otros o para obtener soporte técnico – como con los amables colegas que encontraras en el canal de IRC o la lista de correo de Django.

- Debajo del boton "Share this traceback on a public Web site" (comparte esta traza de error en un sitio publico) puedes hacer clic en el boton, para postear la traza en un sitio público como http://www.dpaste.com/, donde podras pegarlo a una URL, cada vez que decidas compartirlo con otras personas.

- A continuación, la sección *"Request information"* incluye una gran cantidad de información sobre la petición Web que provocó el error: información GET y POST, valores de las cookies y meta información, así como las cabeceras CGI. Él *apéndice G* contiene una referencia completa sobre la información que contienen todos los objetos peticiones.

- Más abajo, en la sección *"Settings"* se encuentra la lista de configuraciones de la instalación de Django en particular. (El cual mencionamos en ROOT_URLCONF) y mencionaremos a lo largo del libro. Él *apéndice D*, cubre en detalle todos los ajustes de configuración disponibles. Por ahora, sólo mira los ajustes para obtener una idea de la información disponible.

La página de error de Django es capaz de mostrar más información en ciertos casos especiales, como por ejemplo, en el caso de error de sintaxis en las plantillas. Lo abordaremos más tarde, cuando discutamos el sistema de plantillas de Django. Por ahora, quita el comentario en la línea offset = int(offset) para que la función de vista funcione de nuevo, normalmente.

¿Eres el tipo de programador al que le gusta depurar con la ayuda de sentencias print cuidadosamente colocadas? Puedes usar la página de error de Django para hacer eso – sin usar la sentencia print. En cualquier lugar de una vista, temporalmente puedes insertar un assert False para provocar una página de error. Luego, podras ver las variables locales y el estado del programa. (Hay maneras más avanzadas de depurar las vistas en Django, lo explicaremos más adelante, pero esta es la forma más rápida y fácil).

Mira el siguiente ejemplo:

```
def horas_adelante(request, offset):

    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt=datetime.datetime.now()+datetime.timedelta(hours=offset)
    assert False
    html = "<html><body><h1>En %s hora(s), seran:</h1><h3>
        %s</h3></body></html>" % (offset, dt)
    return HttpResponse(html)
```

Finalmente, es obvio que la mayor parte de la información mostrada es delicada – ya que expone las entrañas del código fuente de Python, así como también la configuración de Django y sería una estupidez mostrarla al público en Internet. Una persona con malas intenciones podría usar esto para intentar aplicar ingeniería inversa en la aplicación Web y hacer cosas maliciosas.

Por esta razón, la página de error es mostrada sólo cuando el proyecto está en modo depuración. Explicaremos cómo desactivar este modo más adelante. Por ahora, hay que tener en claro que todos los proyectos de Django están en modo depuración automáticamente cuando son creados. (¿Suena familiar? Los errores "Page not found", descriptos en la sección "Errores 404", trabajan de manera similar.)

¿Qué sigue?

Hasta ahora hemos producido las vistas mediante código HTML dentro del código Python. Desafortunadamente, esto casi siempre es una mala idea.

Pero por suerte, con Django podemos hacer esto con un potente motor de plantillas que nos permite separar el diseño de las páginas del código fuente subyacente. Nos sumergiremos en el motor de plantillas de Django en el *próximo capitulo*

CAPÍTULO 4

El sistema de plantillas

En el capítulo anterior, quizás notaste algo extraño en la forma en cómo retornamos el texto en nuestras vistas de ejemplos. Ya que el HTML fue codificado[1] directamente en nuestro código Python, así:

```
def fecha_actual(request):
    ahora = datetime.datetime.now()
    html = <html><body><h1>Fecha:</h1><h3>%s<h/3></body>
        </html>" % ahora
    return HttpResponse(html)
```

Aunque esta técnica fue conveniente para explicar la forma en que trabajan las vistas, no es buena idea codificar, mezclar e incrustar directamente el HTML en las vistas, ya que este convenio conduce a problemas severos:

- Cualquier cambio en el diseño de la página requiere un cambio en el código de Python. El diseño de un sitio tiende a cambiar más frecuentemente que el código de Python subyacente, por lo que sería conveniente si el diseño podría ser cambiado sin la necesidad de modificar el código Python.

- Escribir código Python y diseñar HTML son dos disciplinas diferentes, y la mayoría de los entornos de desarrollo web profesional dividen estas responsabilidades entre personas separadas (o incluso en departamento separados). Diseñadores y programadores HTML/CSS no deberían tener que editar código Python para conseguir hacer su trabajo; ellos deberían tratar con HTML.

- Asimismo, esto es más eficiente si los programadores pueden trabajar sobre el código Python y los diseñadores sobre las plantillas al mismo tiempo, más bien que una persona espere por otra a que termine de editar un solo archivo que contiene ambos: Python y HTML.

Por esas razones, es mucho más limpio y mantenible separar el diseño de la página del código Python en sí mismo. Podemos hacer esto con *el sistema de plantillas* de Django, el cual trataremos en este capítulo.

Sistema básico de plantillas

Una plantilla de Django es una cadena de texto que pretende separar la presentación de un documento de sus datos. Una plantilla define rellenos y diversos bits de lógica

[1] N. del T.: hard-coded:(Codificado en duro)

básica (esto es, etiquetas de plantillas) que regulan cómo debe ser mostrado el documento. Normalmente, las plantillas son usadas para producir HTML, pero las plantillas de Django son igualmente capaces de generar cualquier formato basado en texto.

Comencemos con una simple plantilla de ejemplo. Esta plantilla en Django, describe una página HTML que agradece a una persona por hacer un pedido de una empresa. Piensa en este como un modelo de carta:

```
<html>
<head><title>Orden de pedido</title></head>
<body>

<h1>Orden de pedido</h1>

<p>Estimado: {{ nombre }},</p>

<p>Gracias por el pedido que ordeno de la {{ empresa }}.
El pedido junto con la mercancía se enviaran el
{{ fecha|date:"F j, Y" }}.</p>

<p>Esta es la lista de productos que usted ordeno:</p>

<ul>
{% for pedido in lista_pedido %}
   <li>{{ pedido }}</li>
{% endfor %}
</ul>

{% if garantía %}
   <p>La garantía será incluida en el paquete.</p>
   {% else %}
      <p>Lamentablemente no ordeno una garantía, por lo
      que los daños al producto corren por su cuenta.</p>
{% endif %}

<p>Sinceramente <br /> {{ empresa }}</p>
</body>
</html>
```

Esta plantilla es un archivo HTML básico con algunas variables y etiquetas de plantillas agregadas. Veamos paso a paso, como está construida:

- Cualquier texto encerrado por un par de llaves (por ej. {{ nombre }}) es una *variable*. Esto significa "insertar el valor de la variable a la que se dio ese nombre". ¿Cómo especificamos el valor de las variables?. Vamos a llegar a eso en un momento.
- Cualquier texto que esté rodeado por llaves y signos de porcentaje (por ej. {% if garantía %}) es una *etiqueta de plantilla*. La definición de etiqueta es bastante amplia: una etiqueta sólo le indica al sistema de plantilla "haz algo".

- Este ejemplo de plantilla contiene dos etiquetas: la etiqueta {% for pedido in lista_pedido %} (una etiqueta for) y la etiqueta {% if garantía %} (una etiqueta if).

Una etiqueta for actúa como un simple constructor de bucle, dejándote recorrer a través de cada uno de los ítems de una secuencia. Una etiqueta if, como quizás esperabas, actúa como una cláusula lógica "if". En este caso en particular, la etiqueta comprueba si el valor de la variable garantía se evalúa como True. Si lo hace, el sistema de plantillas mostrará todo lo que hay entre {% if garantía %} y {% endif %}. Si no, el sistema de plantillas no mostrará esto. Nota que la etiqueta {% else %} es opcional.

- Finalmente, el segundo párrafo de esta plantilla, muestra un ejemplo de un *filtro*, con el cual puedes alterar la exposición de una variable. En este ejemplo, {{ fecha|date:"F j, Y" }}, estamos pasando la variable fecha por el filtro date, pasando los argumentos "F j, Y" al filtro. El filtro date formatea fechas en el formato dado, especificado por ese argumento. Los filtros se encadenan mediante el uso de un carácter pipe (|), como una referencia a las tuberías de Unix.

Cada plantilla de Django tiene acceso a varias etiquetas y filtros incorporados, algunos de los cuales serán tratados en las secciones siguientes. El *apéndice E* contiene la lista completa de etiquetas y filtros, es una buena idea familiarizarse con estas etiquetas y filtros, para aprender a usarlos e incorporarlos en tus propios proyectos. También es posible crear tus propios filtros y etiquetas, los cuales cubriremos en el *capítulo 9*.

Usando el sistema de plantillas

Sumerjámonos por un rato en el sistema de plantillas, para entender la forma en que trabajan –por ahora *no* las integraremos en las vistas que creamos en el capítulo anterior.

El objetivo será mostrar cómo trabaja el sistema de plantillas, independientemente del resto de Django (Veámoslo de otra forma: normalmente usaríamos el sistema de plantillas dentro de una vista, sin embargo lo que queremos dejar muy en claro, es que el sistema de plantillas es solo una librería de código Python, que se puede utilizar en *cualquier* parte, no solo en las vista de Django.)

Esta es la forma básica, en la que podemos usar el sistema de plantillas de Django en código Python.

1. Crea un objeto Template pasándole el código en crudo de la plantilla como una cadena.

2. Llama al método render() del objeto Template con un conjunto de variables (o sea, el *contexto*). Este retorna una plantilla totalmente renderizada como una cadena de caracteres, con todas las variables y etiquetas de bloques evaluadas de acuerdo al contexto.

Usando código, esta es la forma que podría verse, solo inicia el intérprete interactivo con: python manage.py shell:

```
>>> from __future__ import print_function
>>> from django import template
>>> t = template.Template('Mi nombre es {{ nombre }}.')
>>> c = template.Context({'nombre': 'Adrian'})
>>> print (t.render(c))
Mi nombre es Adrian.
>>> c = template.Context({'nombre': 'Fred'})
```

```
>>> print (t.render(c))
Mi nombre es Fred.
```

Las siguientes secciones describen cada uno de los pasos con mayor detalle.

Creación de objetos Template

La manera sencilla de crear objetos Template es instanciarlos directamente. La clase Template se encuentra en el módulo django.template, y el constructor toma un argumento, el código en crudo de la plantilla. Vamos a sumergirnos en el intérprete interactivo de Python para ver cómo funciona este código.

En el directorio del proyecto misitio, que creamos con el comando django-admin.py startproject (Cubierto en el *capítulo 2*) tipea: python manage.py shell para iniciar el intérprete interactivo.

UN INTÉRPRETE DE PYTHON ESPECIAL

Si has usado Python antes, tal vez te sorprenda que ejecutemos python manage.py shell en lugar de solo python que inicia el interprete interactivo, pero debemos decirte que el comando manage.py shell tiene una importante diferencia: antes de iniciar el interprete, le pregunta a Django cual archivo de configuraciones usar, el cual incluye ajustes, como la ruta al sistema de plantillas, sin estos ajustes no podrás usarlo, a menos que los importes manualmente.

Si eres curioso, esta es la forma en que trabaja Django tras bastidores. Primero busca la variable de entorno llamada DJANGO_SETTINGS_MODULE, la cual debería encontrarse en la ruta de importación del archivo settings.py. Por ejemplo, puede ser DJANGO_SETTINGS_MODULE o 'misitio.settings', asumiendo que misitio este en la ruta de búsqueda de Python (Python path).

Cuando ejecutas manage.py shell, el comando se encarga de configurar DJANGO_SETTINGS_MODULE por ti. Es por ello que te animamos a usar manage.py shell, en estos ejemplos a fin de reducir la cantidad de ajustes y configuraciones que tengas que hacer.

Django también puede usar IPython o bpython , si están instalados, para iniciar un intérprete interactivo mejorado, el cual agrega funcionalidades extras al simple interprete interactivo plano por defecto.

Si tienes instalados ambos, y quieres elegir entre usar IPython o bpython como intérprete, necesitas especificarlo con la opción -i o --interface de esta forma:

iPython:

```
django-admin.py shell -i ipython
django-admin.py shell --interface ipython
```

bpython:

```
django-admin.py shell -i bpython
django-admin.py shell --interface bpython
```

Para forzar al intérprete a usar el interprete interactivo “plano” usa:

```
django-admin.py shell –plain
```

Comencemos con algunos fundamentos básicos del sistema de plantillas:

```
>>> from __future__ import print_function
>>> from django.template import Template
>>> t = Template('Mi nombre es {{ nombre }}.')
>>> print (t)
```

Si lo estás siguiendo interactivamente, verás algo como esto:

```
<django.template.Template object at 0xb7d5f24c>
```

Ese 0xb7d5f24c será distinto cada vez, y realmente no importa; es la forma simple en que Python "identifica" un objeto de Template.

Cuando creas un objeto Template, el sistema de plantillas compila el código en crudo a uno interno, de forma optimizada, listo para renderizar. Pero si tu código de plantilla incluye errores de tipo sintaxis, la llamada a Template() causará una excepción TemplateSyntaxError:

```
>>> from __future__ import print_function
>>> from django.template import Template
>>> t = Template('{% notatag %} ')
```

El término "block tag" "etiqueta de bloque" hace referencia a {% notatag %}. "Etiqueta de plantilla" y "bloque de plantilla" son sinónimos.

El sistema lanza una excepción TemplateSyntaxError por alguno de los siguientes casos:

- Bloques de etiquetas inválidos
- Argumentos inválidos para una etiqueta válida
- Filtros inválidos
- Argumentos inválidos para filtros válidos
- Sintaxis inválida de plantilla
- Etiquetas de bloque sin cerrar (para etiquetas de bloque que requieran la etiqueta de cierre)

Renderizar una plantilla

Una vez que tienes un objeto Template, le puedes pasar datos brindandole un *contexto*. Un contexto es simplemente un conjunto de variables y sus valores asociados. Una plantilla usa estas variables para llenar y evaluar estas etiquetas de bloque.

Un contexto es representado en Django por la clase Context, ésta se encuentra en el módulo django.template. Su constructor toma un argumento opcional: un diccionario que mapea nombres de variables con valores. Llama al método render() del objeto Template con el contexto para "llenar" la plantilla:

```
>>> from __future__ import print_function
>>> from django.template import Context, Template
>>> t = Template("Mi nombre es {{ nombre }}.")
>>> c = Context({"nombre": "Estefanía"})
>>> t.render(c)
'Mi nombre es Estefanía.'
```

Una cosa que debemos apuntar aquí, es que el valor de retorno de t.render(c) es un objeto unicode –No una cadena normal de Python–. Como sabes podemos usar la

"u" al inicio de la cadena para usar objetos unicode en Python 2, sin embargo en python3 esto no es necesario, ya que soporta nativamente objetos unicode. Sin embargo no está de más decirte que Django también soporta nativamente datos unicode en lugar de cadenas normales en todo el framework. Si comprendes las repercusiones de esto, estarás agradecido por las cosas sofisticadas que hace Django tras bastidores, para facilitarte el trabajo.

Si no las comprendes, no te preocupes por ahora; solo debes saber que Django hace que el soporte unicode sea indoloro para tus aplicaciones, para que puedan soportar una gran variedad de caracteres, que van más allá del básico "A-Z" del idioma ingles.

DICCIONARIOS Y CONTEXTOS

Un diccionario en Python es un mapeo entre llaves conocidas y valores de variables.

Un Context (contexto) es similar a un diccionario, pero un Context provee funcionalidades adicionales, como se cubre en el *capítulo 9*.

Los nombres de las variables deben comenzar con una letra (A-Z o a-z) y pueden contener dígitos, guiones bajos y puntos. (Los puntos son un caso especial al que llegaremos en un momento). Los nombres de variables son sensibles a mayúsculas-minúsculas.

Este es un ejemplo de compilación y renderización de una plantilla, usando la plantilla de muestra del comienzo de este capítulo:

```
>>> from __future__ import print_function
>>> from django.template import Template, Context
>>> raw_template = """<p>Estimado: {{ nombre }},</p>
...
... <p>Gracias por el pedido que ordeno de {{ empresa }}. El
... pedido se enviara el {{ ship_date|date: "j F Y" }}.</p>
...
... {% if garantía %}
... <p>La garantía será incluida en el paquete.</p>
... {% else %}
... <p>Lamentablemente no ordeno una garantía, por lo que los
... daños al producto corren por su cuenta.</p>
... {% endif %}
...
... <p>Sinceramente <br />{{ empresa }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'nombre': 'Juan Pérez',
... 'empresa': 'Entrega veloz',
... 'fecha': datetime.date(2014, 10, 10),
... 'ordered_warranty': False})
>>> t.render(c)
u"<p>Estimado Juan Pérez,</p>\n\n<p>Gracias por el pedido que ordeno de Entrega
veloz. El pedido se enviara el 10 Octubre 2014.</p>\n\n\n<p> Lamentablemente no
ordeno una garantía, por lo que los daños\nal producto corren por su cuenta.</p>\n\n\n
<p>Sinceramente,<br />Entrega veloz</p>"
```

Veamos paso a paso este código, una sentencia a la vez:

- Primero, importamos la clase Template y Context, ambas se encuentran en el módulo django.template. Guardamos en texto crudo, nuestra plantilla en la

variable raw_template. Nota que usamos triple comillas para delimitar la cadena de caracteres, debido a que abarca varias líneas; en el código Python, las cadenas de caracteres delimitadas con una sola comilla indican que no puede abarcar varias líneas.

- Luego, creamos un objeto plantilla, t, pasándole raw_template al constructor de la clase Template.

- Importamos el módulo datetime desde la biblioteca estándar de Python, porque lo vamos a necesitar en la próxima sentencia.
- Entonces, creamos un objeto Context , c. El constructor de Context toma un diccionario de Python, el cual mapea los nombres de las variables con los valores. Aquí, por ejemplo, especificamos que nombre es 'Juan Pérez', empresa es 'Entrega Veloz', y así sucesivamente.

- Finalmente, llamamos al método render() sobre nuestro objeto de plantilla, pasando a éste el contexto. Este retorna la plantilla renderizada – esto es, reemplaza las variables de la plantilla con los valores reales de las variables, y ejecuta cualquier bloque de etiquetas.

  Nota que el párrafo "Lamentablemente no ordeno una garantía" fue mostrado porque la variable garantía se evalúa como False. También nota que la fecha 10 Octubre 2014, es mostrada acorde al formato de cadena de caracteres j F Y. (Explicaremos los formatos de cadenas de caracteres para el filtro date a la brevedad).

  Si eres nuevo en Python, quizás te preguntes por qué la salida incluye los caracteres de nueva línea ('\n') en vez de mostrar los saltos de línea. Esto sucede porque es una sutileza del intérprete interactivo de Python: la llamada a t.render(c) retorna una cadena de caracteres, y el intérprete interactivo, por omisión, muestra una *representación* de ésta, en vez de imprimir el valor de la cadena. Si quieres ver la cadena de caracteres con los saltos de líneas como verdaderos saltos de líneas en vez de caracteres '\n', usa la sentencia print: print (t.render(c)).

Estos son los fundamentos del uso del sistema de plantillas de Django: sólo escribe una plantilla, crea un objeto Template, crea un Context, y llama al método render().

Múltiples contextos, mismas plantillas

Una vez que tengas un objeto Template, puedes renderizarlo con múltiples contextos, por ejemplo:

```
>>> from __future__ import print_function
>>> from django.template import Template, Context
>>> t = Template('Hola, {{ nombre }}')
>>> print (t.render(Context({'nombre': 'Juan'})))
Hola, Juan
>>> print (t.render(Context({'nombre': 'Julia'})))
Hola, Julia
>>> print (t.render(Context({'nombre': 'Paty'})))
Hola, Paty
```

Cuando estés usando la misma plantilla fuente para renderizar múltiples contextos como este, es más eficiente crear el objeto Template *una sola vez* y luego llamar al método render() sobre éste muchas veces:

```
# Mal
for nombre in ('Juan', 'Julia', 'Paty'):
    t = Template('Hola, {{ nombre }}')
        print (t.render(Context({'nombre': nombre})))

# Bien
t = Template('Hola, {{ nombre }}')
for nombre in ('Juan', 'Julia', 'Paty'):
    print (t.render(Context({'nombre': nombre})))
```

El analizador sintáctico de las plantillas de Django es bastante rápido. Detrás de escena, la mayoría de los analizadores pasan con una simple llamada a una expresión regular corta. Esto es un claro contraste con el motor de plantillas de XML, que incurre en la excesiva actividad de un analizador XML, y tiende a ser órdenes de magnitud más lento que el motor de renderizado de Django.

Búsqueda del contexto de una variable

En los ejemplos vistos hasta el momento, pasamos valores simples a los contextos – en su mayoría cadena de caracteres, más un *datetime.date*. Sin embargo, el sistema de plantillas maneja elegantemente estructuras de datos más complicadas, como listas, diccionarios y objetos personalizados.

La clave para recorrer estructuras de datos complejos en las plantillas de Django es el uso del carácter punto (.), usa un punto para acceder a las claves de un diccionario, atributos, índices o métodos de un objeto.

Esto es mejor ilustrarlos con algunos ejemplos. Por ejemplo, imagina que pasas un diccionario de Python a una plantilla. Para acceder al valor de ese diccionario por su clave, solo usa el punto:

```
>>> from django.template import Template, Context
>>> persona = {'nombre': 'Silvia', 'edad': '43'}
>>> t = Template('{{ persona.nombre }} tiene
      {{ persona.edad }} años.')
>>> c = Context({'persona': persona})
>>> t.render(c)
u'Silvia tiene 43 años.'
```

De forma similar, los puntos te permiten acceder a los atributos de los objetos. Por ejemplo, un objeto de Python *datetime.date* tiene los atributos *year, month* y *day,* y puedes usar el punto para acceder a ellos en las plantillas de Django:

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
```

```
>>> d.day
2
>>> t = Template('El mes es {{ date.month }} y el año
    es {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
u'El mes es 5 y el años es 1993.'
```

Este ejemplo usa una clase personalizada, que demuestra que la variable punto permite acceder a objetos de forma arbitraria:

```
>>> from django.template import Template, Context
>>> class Persona(object):
... def __init__(self, nombre, apellido):
... self.nombre, self.apellido = nombre, apellido
>>> t = Template('Hola, {{ persona.nombre }}
    {{ persona.apellido }}.')
>>> c = Context({'persona': Persona('Juan', 'Pérez')})
>>> t.render(c)
u'Hola, Juan Pérez.'
```

Los puntos también son utilizados para llamar a métodos sobre los objetos. Por ejemplo, cada cadena de caracteres de Python posee métodos upper() y isdigit(), por lo que puedes llamar a estos métodos en las plantillas de Django usando la misma sintaxis de punto:

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} –
    {{ var.isdigit }}')
>>> t.render(Context({'var': 'hola'}))
u'hola -- HOLA -- False'
>>> t.render(Context({'var': '123'}))
u'123 -- 123 -- True'
```

Observa que no tienes que incluir los paréntesis en las llamadas a los métodos. Además, tampoco es posible pasar argumentos a los métodos; sólo puedes llamar a los métodos que no requieran argumentos. (Explicaremos esta filosofía, mas adelante en este capítulo).

Finalmente, los puntos también son usados para acceder a los índices de las listas, por ejemplo:

```
>>> from django.template import Template, Context
>>> t = Template('Fruta 2 es {{ frutas.2 }}.')
>>> c = Context({'frutas': ['manzana', 'plátano', 'pera']})
>>> t.render(c)
u'Fruta 2 es pera.'
```

Los índices negativos de las listas no están permitidos. Por ejemplo, la variable {{ frutas. -1 }} causará una TemplateSyntaxError.

LISTAS EN PYTHON

Las listas de Python comienzan en cero, entonces el primer elemento es el 0, el segundo es el 1 y así sucesivamente.

La búsqueda del punto puede resumirse a esto: cuando el sistema de plantillas encuentra un punto en una variable, intentara buscar en este orden:

1. Diccionario (por ej. foo["bar"])
2. Atributo (por ej. foo.bar)
3. Llamada de método (por ej. foo.bar())
4. Índice de lista (por ej. foo[bar])

El sistema utiliza el primer tipo de búsqueda que funcione. Es la lógica de cortocircuito.

Los puntos pueden ser anidados a múltiples niveles de profundidad. El siguiente ejemplo usa {{ persona.name.upper }}, el que se traduce en una búsqueda de diccionario (persona['nombre']) y luego en una llamada a un método (upper()):

```
>>> from django.template import Template, Context
>>> persona = {'nombre': 'Silvia', 'edad': '43'}
>>> t = Template('{{ persona.nombre.upper }} tiene
   {{ person.age }} años.')
>>> c = Context({'persona': persona})
>>> t.render(c)
u'SILVIA tiene 43 años.'
```

Comportamiento de la llamada a los métodos

La llamada a los métodos es ligeramente más compleja que los otros tipos de búsqueda. Aquí hay algunas cosas a tener en cuenta:

- Si, durante la búsqueda de método, un método provoca una excepción, la excepción será propagada, a menos que la excepción tenga un atributo silent_variable_failure cuyo valor sea True.

- Si la excepción *contiene* el atributo silent_variable_failure, la variable será renderizada como un string vacío, por ejemplo:

```
>>> t = Template("Mi nombre es {{ persona.nombre }}.")
>>> class ClasePersona:
... def nombre(self):
... raise AssertionError, "foo"
>>> p = ClasePersona()
>>> t.render(Context({"persona": p}))
Traceback (most recent call last):
...
AssertionError: foo
>>> class SilentAssertionError(AssertionError):
... silent_variable_failure = True
>>> class ClasePersona2:
... def nombre(self):
... raise SilentAssertionError
>>> p = ClasePersona2()
>>> t.render(Context({"persona": p}))
u'Mi nombre es .'
```

- La llamada a un método funcionará sólo si el método no requiere argumentos. En otro caso, el sistema pasará a la siguiente búsqueda de tipo índice de lista.

- Evidentemente, algunos métodos tienen efectos secundarios, por lo que sería absurdo, en el mejor de los casos, y posiblemente un agujero de seguridad, permitir que el sistema de plantillas tenga acceso a ellos.

  Digamos, por ejemplo, que tienes un objeto CuentaBanco que tiene un método borrar(). Una plantilla no debería permitir incluir algo como {{ cuenta.borrar }}, donde cuenta es un objeto CuentaBanco, ¡ya que el objeto será borrado cuando se renderice la plantilla!

  Para prevenir esto, asigna el atributo alters_data de la función en el método:

```
def delete(self):

    # Borra una cuenta
    delete.alters_data = True
```

El sistema de plantillas no debería ejecutar cualquier método marcado de esta forma. En otras palabras, si una plantilla incluye {{ cuenta.borrar}} y el método borrar(), marcado como alters_data=True, esta etiqueta no ejecutará el método delete(). Ya que este fallará silenciosamente.

¿QUE ES SELF?

Self es simplemente el nombre convencional para el primer argumento de un método en Python.

Por ejemplo, un método definido de la forma meth(self, a, b, c) debe ser llamado con x.meth(a, b, c), por alguna instancia de la clase x, en la cual ocurre la definición; de esta forma el método llamado pensara que es llamado como meth(x, a, b, c).

¿Cómo se manejan las variables inválidas?

De forma predeterminada, si una variable no existe, el sistema de plantillas renderiza esta como una cadena vacía, fallando silenciosamente, por ejemplo:

```
>>> from django.template import Template, Context
>>> t = Template('Tu nombre es {{ nombre }}.')
>>> t.render(Context())
u'Tu nombre es .'
>>> t.render(Context({'var': 'hola'}))
u'Tu nombre es .'
>>> t.render(Context({'NOMBRE': 'hola'}))
u'Tu nombre es .'
>>> t.render(Context({'Nombre': 'hola'}))
u'Tu nombre es .''
```

El sistema falla silenciosamente en vez de levantar una excepción porque intenta ser flexible a los errores humanos. En este caso, todas las búsquedas fallan porque los nombres de las variables, o su capitalización es incorrecta. En el mundo real, es inaceptable para un sitio web ser inaccesible debido a un error de sintaxis tan pequeño.

Jugando con objetos Context

La mayoría de las veces, solo tendrás que instanciar un objeto Context pasándole un diccionario, completamente poblado a Context. Sin embargo, también puedes agregar y quitar elementos de un objeto Context una vez que éste está instanciado, usando la sintaxis estándar de los diccionarios de Python:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
Traceback (most recent call last):
...
KeyError: 'foo'
>>> c['nuevavariable'] = 'hola'
>>> c['nuevavariable']
'hola'
```

Etiquetas básicas de plantillas y filtros

Como hemos mencionamos anteriormente, el sistema de plantillas se distribuye con etiquetas y filtros incorporados. Las secciones que siguen proveen un resumen de la mayoría de las etiquetas y filtros.

Etiquetas

if/else

La etiqueta {% if %} evalúa una variable, y si esta es "true" (esto es, existe, no está vacía y no es un valor Boolean falso), el sistema mostrará todo lo que hay entre {% if %} y {% endif %}, por ejemplo:

```
{% if es_fin_de_semana %}
   <p>¡Bienvenido fin de semana!</p>
{% endif %}
```

La etiqueta {% else %} es opcional:

```
{% if es_fin_de_semana %}
   <p>¡Bienvenido fin de semana!</p>
{% else %}
   <p>De vuelta al trabajo.</p>
{% endif %}
```

LAS "VERDADES" EN PYTHON

En Python y en el sistema de plantillas de Django, los siguientes objetos son evaluados como False (falsos) en un contexto booleano:

- Una lista vacía ([]),
- Una tupla vacía (()),
- Un diccionario vacío ({}),

- Una cadena vacía (''),
- El cero (0),
- El objeto especial None
- El objeto False (obviamente)
- Objetos personalizados que definen su propio comportamiento en un contexto boleano (Es la ventaja de usar Python)
- Todo lo demás es evaluado como verdadero (True).

None: es un valor especial de Python que devuelven funciones que o bien no tienen sentencia de return o bien tienen una sentencia de return sin argumento.

La etiqueta {% if %} acepta and, or, o not para testear múltiples variables, o para negarlas,
por ejemplo:

```
{% if lista_atletas and lista_entrenadores %}
   Atletas y Entrenadores están disponibles
{% endif %}

{% if not lista_atletas %}
   No hay atletas
{% endif %}

{% if lista_atletas or lista_entrenadores %}
   Hay algunos atleta o algunos entrenadores
{% endif %}
{% if not lista_atletas or lista_entrenadores %}
   No hay atletas o no hay entrenadores.
{% endif %}

{% if lista_atletas and not lista_entrenadores %}
   Hay algunos atletas y absolutamente ningún entrenador.
{% endif %}
```

Las etiquetas {% if %} no permiten las cláusulas and y or en la misma etiqueta, porque el orden de evaluación lógico puede ser ambiguo. Por ejemplo, esto es inválido:

```
{% if lista_atletas and lista_entrenadores or lista_porristas %}
```

No se admite el uso de paréntesis para controlar el orden de las operaciones. Si necesitas paréntesis, considera efectuar la lógica en el código de la vista para simplificar las plantillas.
Aún así, si necesitas combinar and y or para hacer lógica avanzada, usa etiquetas {% if %} anidadas, por ejemplo:

```
{% if lista_atletas %}
   {% if lista_entrenadores or lista_porristas %}
      ¡Tenemos atletas y entrenadores o porristas!
   {% endif %}
{% endif %}
```

Usar varias veces el mismo operador lógico está bien, pero no puedes combinar diferentes operadores. Por ejemplo, esto es válido:

```
{% if lista_atletas or lista_entrenadores or lista_padres or lista_maestros %}
```

No hay una etiqueta {% elif %}. En su lugar usa varias etiquetas {% if %} anidadas para conseguir el mismo resultado:

```
{% if lista_atletas %}
   <p>Aquí están los atletas: {{ lista_atletas }}.</p>
{% else %}
   <p>No hay atletas disponibles.</p>
   {% if lista_entrenadores %}
      <p>Aquí están los entrenadores: {{ lista_entrenadores }}.</p>
   {% endif %}
{% endif %}
```

Asegúrate de cerrar cada {% if %} con un {% endif %}. En otro caso, Django levantará la excepción TemplateSyntaxError.

For

La etiqueta {% for %} permite iterar sobre cada uno de los elementos de una secuencia. Como en la sentencia for de Python, la sintaxis es for X in Y, dónde Y es la secuencia sobre la que se hace el bucle y X es el nombre de la variable que se usará para cada uno de los ciclos del bucle.

Cada vez que atravesamos el bucle, el sistema de plantillas renderizará todo entre {% for %} y {% endfor %}.

Por ejemplo, puedes usar lo siguiente para mostrar una lista de atletas tomadas de la variable lista_atletas:

```
<ul>
{% for atleta in lista_atletas %}
   <li>{{ atleta.nombre }}</li>
{% endfor %}
</ul>
```

Agrega reversed a la etiqueta para iterar sobre la lista en orden inverso:

```
{% for atleta in lista_atletas reversed %}
...
{% endfor %}
```

Es posible anidar etiquetas {% for %}:

```
{% for pais in paises %}
   <h1>{{ pais.nombre }}</h1>
   <ul>
   {% for ciudad in pais.lista_ciudades %}
       <li>{{ ciudad }}</li>
   {% endfor %}
   </ul>
{% endfor %}
```

Un uso muy común de la etiqueta for, es para comprobar el tamaño de una lista antes de iterar sobre ella y mostrar algún texto en especial, si la lista está vacía.:

```
{% if lista_atletas %}
   {% for atleta in lista_atletas %}
      <p>{{ atleta.nombre }}</p>
   {% endfor %}
{% else %}
   <p>No hay atletas. Únicamente programadores.</p>
{% endif %}
```

El ejemplo anterior es tan común, que la etiqueta for soporta una clausula opcional:
{% empty %} que te permite definir lo que hay que hacer si la lista está vacía. El siguiente ejemplo es equivalente al anterior:

```
{% for atleta in lista_atletas %}
   <p>{{ athlete.nombre }}</p>
{% empty %}
   <p>No hay atletas. Únicamente programadores.</p>
{% endfor %}
```

No se admite la "ruptura" de un bucle antes de que termine. Si quieres conseguir esto, cambia la variable sobre la que estás iterando para que incluya sólo los valores sobre los cuales quieres iterar. De manera similar, no hay apoyo para la sentencia "continue" que se encargue de retornar inmediatamente al inicio del bucle. (Consulta la sección "Filosofía y limitaciones" más adelante para comprender el razonamiento detrás de esta decisión de diseño.)

Dentro de cada bucle, la etiqueta {% for %} permite acceder a una variable llamada forloop, dentro de la plantilla. Esta variable tiene algunos atributos que toman información acerca del progreso del bucle:

- forloop.counter es siempre asignada a un número entero representando el número de veces que se ha entrado en el bucle. Esta es indexada a partir de 1, por lo que la primera vez que se ingresa al bucle, forloop.counter será 1. Aquí un ejemplo:

  ```
  {% for objeto in lista %}
     <p>{{ forloop.counter }}: {{ objeto }}</p>
  {% endfor %}
  ```

- forloop.counter0 es como forloop.counter, excepto que esta es indexada a partir de cero. Contendrá el valor 0 la primera vez que se atraviese el bucle.

- forloop.revcounter es siempre asignado a un entero que representa el número de iteraciones que faltan para terminar el bucle. La primera vez que se ejecuta el bucle forloop.revcounter será igual al número de elementos que hay en la secuencia. La última vez que se atraviese el bucle, a forloop.revcounter se la asignará el valor 1.

- forloop.revcounter0 es como forloop.revcounter, a excepción de que está indexada a partir de cero. La primera vez que se atraviesa el bucle, forloop.revcounter0 es asignada al número de elementos que hay en la secuencia menos 1. La última vez que se atraviese el bucle, el valor de esta será 0.

- forloop.first es un valor booleano asignado a True si es la primera vez que se pasa por el bucle. Esto es conveniente para ocasiones especiales:

```
{% for objeto in objetos %}
   {% if forloop.first %}<li class="first">{% else %}<li>{% endif %}
      {{ objeto }} </li>
{% endfor %}
```

- forloop.last es un valor booleano asignado a True si es la última pasada por el bucle. Un uso común es para esto es poner un carácter pipe entre una lista de enlaces:

```
{% for enlace in enlaces %} {{ enlace }} {% if not forloop.last %} | {% endif %}
{% endfor %}
```

  El código de la plantilla de arriba puede mostrar algo parecido a esto:

```
Enlace1 | Enlace2 | Enlace3 | Enlace4
```

  También se usa comúnmente, para poner comas entre palabras de una lista: por ejemplo para mostrar una lista de lugares favoritos:

```
{% for p in lugares %}{{ p }}{% if not forloop.last %},
{% endif %}{% endfor %}
```

- forloop.parentloop hace referencia al objeto *padre* de forloop, en el caso de bucles anidados. Aquí un ejemplo:

```
{% for pais in paises %}
   <table>
   {% for ciudad in pais.lista_ciudades %}
      <tr>
        <td>pais #{{ forloop.parentloop.counter }}</td>
         <td>City #{{ forloop.counter }}</td>
         <td>{{ ciudad }}</td>
      </tr>
   {% endfor %}
   </table>
{% endfor %}
```

La variable mágica forloop está únicamente disponible dentro del bucle. Después de que el analizados sintáctico encuentra {% endfor %}, forloop desaparece.

CONTEXTOS Y LA VARIABLE FORLOOP

Dentro de un bloque {% for %}, las variables existentes se mueven fuera del bloque a fin de evitar sobrescribir la variable mágica forloop. Django expone este contexto moviendo forloop.parentloop. Generalmente no necesitas preocuparte por esto, si provees una variable a la plantilla llamada forloop (a pesar de que no lo recomendamos), se llamará forloop.parentloop mientras esté dentro del bloque {% for %}.

ifequal/ifnotequal

El sistema de plantillas de Django a propósito no es un lenguaje de programación completo y por lo tanto no permite ejecutar sentencias arbitrarias de Python. (Más sobre esta idea en la sección "Filosofía y limitaciones"). Sin embargo, es bastante común que una plantilla requiera comparar dos valores y mostrar algo si ellos son iguales – Django provee la etiqueta
{% ifequal %} para este propósito.

La etiqueta {% ifequal %} compara dos valores y muestra todo lo que se encuentra entre {% ifequal %} y {% endifequal %} si el valor es igual.

Este ejemplo compara las variables usuario y actual_usuario de la plantilla:

```
{% ifequal usuario actual_usuario %}
   <h1>¡Bienvenido!</h1>
{% endifequal %}
```

Los argumentos pueden ser strings "hard-codeados", con comillas simples o dobles, por lo que lo siguiente es válido:

```
{% ifequal seccion 'noticias' %}
   <h1>Noticias</h1>
{% endifequal %}

{% ifequal seccion "comunidad" %}
   <h1>Comunidad</h1>
{% endifequal %}
```

Tal como {% if %}, la etiqueta {% ifequal %} admite opcionalmente la etiqueta
{% else %}:

```
{% ifequal seccion 'noticias' %}
   <h1>Noticias</h1>
{% else %}
   <h1>No hay noticias nuevas</h1>
{% endifequal %}
```

En las variables de plantilla, únicamente las cadenas de texto, enteros y los números decimales son permitidos como argumentos para {% ifequal %}. Estos son ejemplos válidos:

```
{% ifequal variable 1 %}
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

Cualquier otro tipo de variables, tales como diccionarios de Python, listas, o booleanos, no pueden ser comparadas con {% ifequal %}. Estos ejemplos son inválidos:

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

Si necesitas comprobar cuando algo es verdadero o falso, usa la etiqueta {% if %} en lugar de {% ifequal %}.

Comentarios

Al igual que en HTML o en un lenguaje de programación como Python, el lenguaje de plantillas de Django permite usar comentarios. Para designar un comentario, usa {# #}:

```
{# Esto es un comentario #}
```

Este comentario no será mostrado cuando la plantilla sea renderizada.

Un comentario no puede abarcar múltiples líneas. Esta limitación mejora la performance del analizador sintáctico de plantillas. En la siguiente plantilla, la salida del renderizado mostraría exactamente lo mismo que la plantilla (esto es, la etiqueta comentario no será tomada como comentario):

```
Esto es una {# Esto no es
un comentario #}
prueba.
```

Si quieres usar un comentario que abarque varias líneas, usa la etiqueta {% comment %}, así:

```
{% comment %}
  Este es un comentario
  que abarca varias líneas
{% endcomment %}
```

Filtros

Como explicamos anteriormente en este capítulo, los filtros de plantillas son formas simples de alterar el valor de una variable antes de mostrarla. Los filtros se parecen a esto:

```
{{ nonmbre|lower }}
```

Esto muestra el valor de {{ nombre }} después de aplicarle el filtro lower, el cual convierte el texto a minúscula. Usa una pipe o tubería (|) para aplicar el filtro.

Los filtros pueden ser *encadenados* – esto quiere decir que, la salida de uno de los filtros puede ser aplicada al próximo–. Aquí un ejemplo que toma el primer elemento de una lista y la convierte a mayusculas:

```
{{ mi_lista|first|upper }}
```

Algunos filtros toman argumentos. Un filtro con argumentos se ve de este modo:

```
{{ bio|truncatewords:"30" }}
```

Esto muestra las primeras 30 palabras de la variable bio. Los argumentos de los filtros están siempre entre comillas dobles.

Los siguientes son algunos de los filtros más importantes; el Apéndice E cubre el resto.

- addslashes: Agrega una contra-barra antes de cualquier contra-barra, comilla simple o comilla doble. Esto es útil si el texto producido está incluido en un string de JavaScript.

- date: Formatea un objeto date o datetime de acuerdo al formato tomado como parámetro, por ejemplo:

  ```
  {{ fecha|date:"F j, Y" }}
  ```

  El formato de los strings está definido en el Apéndice E.

- escape: Escapa ampersands(&), comillas, y corchetes del string tomado. Esto es usado para desinfectar datos suministrados por el usuario y asegurar que los datos son válidos para XML y XHTML. Específicamente, escape hace estas conversiones:

  - Convierte & en &
  - Convierte < en <
  - Convierte > en >
  - Convierte " (comilla doble) en "
  - Convierte ' (comilla simple) en '

- length: Retorna la longitud del valor. Puedes usar este con una lista o con un string, o con cualquier objeto Python que sepa como determinar su longitud (o sea cualquier objeto que tenga el método __len__()).

Filosofía y Limitaciones

Ahora que tienes una idea del lenguaje de plantillas de Django, debemos señalar algunas de sus limitaciones intencionales, junto con algunas filosofías detrás de la forma en que este funciona.

Más que cualquier otro componente de la aplicación web, las opiniones de los programadores sobre el sistema de plantillas varía extremadamente. El hecho de que Python sólo implemente decenas, sino cientos, de lenguajes de plantillas de código abierto lo dice todo. Cada uno fue creado probablemente porque su desarrollador estima que todos los existentes son inadecuados. (¡De hecho, se dice que es un rito para los desarrolladores de Python escribir su propio lenguaje de plantillas! Si todavía no lo has hecho, tenlo en cuenta. Es un ejercicio divertido).

Con eso en la cabeza, debes estar interesado en saber que Django no requiere que uses su lenguaje de plantillas. Pero Django pretende ser un completo framework que provee todas las piezas necesarias para que el desarrollo web sea productivo, quizás a veces es más conveniente usar el sistema de plantillas de Django que otras bibliotecas de plantillas de Python, pero no es un requerimiento estricto en ningún sentido. Como verás en la próxima sección "Uso de plantillas en las vistas", es muy fácil usar otro lenguaje de plantillas con Django.

Aún así, es claro que tenemos una fuerte preferencia por el sistema de plantillas de Django. El sistema de plantillas tiene raíces en la forma en que el desarrollo web se realiza en World Online y la experiencia combinada de los creadores de Django. Éstas son algunas de esas filosofías:

- **La lógica de negocios debe ser separada de la presentación lógicas** Vemos al sistema de plantillas como una herramienta que controla la presentación y la lógica relacionado a esta – y eso es todo. El sistema de plantillas no debería admitir funcionalidad que vaya más allá de este concepto básico.

  Por esta razón, es imposible llamar a código Python directamente dentro de las plantillas de Django. Todo "programador" está fundamentalmente

limitado al alcance de lo que una etiqueta puede hacer. Es posible escribir etiquetas personalizadas que hagan cosas arbitrarias, pero las etiquetas de Django intencionalmente no permiten ejecutar código arbitrario de Python.

- **La sintaxis debe ser independiente de HTML/XML**. Aunque el sistema de plantillas de Django es usado principalmente para producir HTML, este pretende ser útil para formatos no HTML, como texto plano. Algunos otros lenguajes de plantillas están basados en XML, poniendo toda la lógica de plantilla con etiquetas XML o atributos, pero Django evita deliberadamente esta limitación. Requerir un XML válido para escribir plantillas introduce un mundo de errores humanos y mensajes difícil de entender, y usando un motor de XML para parsear plantillas implica un inaceptable nivel de overhead en el procesamiento de la plantilla.

- **Los diseñadores se supone que se sienten más cómodos con el código HTML**. El sistema de plantillas no está diseñado para que las plantillas necesariamente sean mostradas de forma agradable en los editores WYSIWYG tales como Dreamweaver. Eso es también una limitación severa y no permitiría que la sintaxis sea tan clara como lo es. Django espera las plantillas de los autores para estar cómodo editando HTML directamente.

- **Se supone que los diseñadores no son programadores Python.** El sistema de plantillas de los autores reconoce que las plantillas de las páginas web son en la mayoría de los casos escritos por diseñadores, no por programadores, y por esto no debería asumir ningún conocimiento de Python.

  Sin embargo, el sistema también pretende acomodar pequeños grupos en los cuales las plantillas sean creadas por programadores de Python. Esto ofrece otro camino para extender la sintaxis del sistema escribiendo código Python puro. (Más de esto en el capítulo 9).

- **No se pretende inventar un lenguaje de programación**. El objetivo es ofrecer sólo la suficiente funcionalidad de programación, tales como ramificación e iteración, que son esenciales para hacer presentaciones relacionadas a decisiones.

Como resultado de esta filosofía, el lenguaje de plantillas de Django tiene las siguientes limitaciones:

- *Una plantilla no puede asignar una variable o cambiar el valor de esta*. Esto es posible escribiendo una etiqueta personalizada para cumplir con esta meta (ve el capítulo *10*), pero la pila de etiquetas de Django no lo permite.

- *Una plantilla no puede llamar código Python crudo*. No hay forma de ingresar en "modo Python" o usar sentencias puras de Python. De nuevo, esto es posible creando plantillas personalizadas, pero la pila de etiquetas de Django no lo permite.

Usando el sistema de plantillas en las vistas

Has aprendido el uso básico del sistema de plantillas; ahora vamos a usar este conocimiento para crear una vista. Recordemos la vista fecha_actual en misitio.views, la que comenzamos en el capítulo anterior. Se veía como esta:

```
from django.http import HttpResponse
import datetime

def fecha_actual(request):
    ahora= datetime.datetime.now()
    html = "<html><body>Hoy es: %s.</body></html>" % ahora
    return HttpResponse(html)
```

Vamos a cambiar esta vista usando el sistema de plantillas de Django. Primero, podemos pensar en algo como esto:

```
import datetime

from django.template import Template, Context
from django.http import HttpResponse

def fecha_actual(request):
   ahora = datetime.datetime.now()
   t = Template("<html><body>Hoy es {{ fecha_actual }}.</body></html>")
   html = t.render(Context({'fecha_actual': ahora}))
   return HttpResponse(html)
```

Seguro, esta vista usa el sistema de plantillas, pero no soluciona el problema que planteamos en la introducción de este capítulo. A saber, la plantilla sigue estando incrustada en el código Python. Vamos a solucionar esto poniendo la plantilla en un *archivo separado*, que la vista cargará automáticamente.

Puedes considerar primero guardar la plantilla en algún lugar del disco y usar las funcionalidades de Python para abrir y leer el contenido de la plantilla. Esto puede verse así, suponiendo que la plantilla esté guardada en /home/djangouser/templates/miplantilla.html:

```
import datetime

from django.template import Template, Context
from django.http import HttpResponse

def fecha_actual(request):
    ahora = datetime.datetime.now()
    # Manera simple de usar plantillas del sistema de archivos.
    # Esto es malo, porque no toma en cuenta los archivos no encontrados.
    fp = open('/home/djangouser/templates/miplantilla.html')
    t = Template(fp.read())
    fp.close()
    html = t.render(Context({'fecha_actual': ahora}))
    return HttpResponse(html)
```

Esta aproximación, sin embargo, es poco elegante por estas razones:

- No maneja el caso en que no encuentre el archivo. Si el archivo mytemplate.html no existe o no es accesible para lectura, la llamada a open() levantará la excepción IOError.

- Involucra la ruta de tu plantilla. Si vas a usar esta técnica para cada una de las funciones de las vistas, estarás duplicando rutas de plantillas. ¡Sin mencionar que esto implica teclear mucho más!

- Incluye una cantidad aburrida de código repetitivo. Tienes mejores cosas para hacer en vez de escribir open(), fp.read() y fp.close() cada vez que cargas una plantilla

Para solucionar estos problemas, usamos *cargadores de plantillas* y *directorios de plantillas*, los cuales son descritos, en las siguientes secciones.

Cargadores de plantillas

Django provee una práctica y poderosa API para cargar plantillas del disco, con el objetivo de quitar la redundancia en la carga de la plantilla y en las mismas plantillas.

Para usar la API para cargar plantillas, primero necesitas indicarle al framework dónde están guardadas tus plantillas. El lugar para hacer esto es en el *archivo de configuración*, que mencionamos en el capítulo anterior, cuando introducimos los ajuste en ROOT_URLCONF (El archivo de configuración de Django es el lugar para poner configuraciones para tu instancia de Django).

Si estas siguiéndonos abre tú archivo settings.py y agrega la variable TEMPLATE_DIRS:

```
TEMPLATE_DIRS = (
    # Pon cadenas del tipo "/home/html/django_templates" o
    # En Windows  usa "C:/www/django/templates".
)
```

Estas configuraciones le indican al mecanismo de carga de plantillas dónde buscar las plantillas. Por omisión, ésta es una tupla vacía. Elige un directorio en el que desees guardar tus plantillas y agrega este a TEMPLATE_DIRS, así:

```
TEMPLATE_DIRS = (
    '/home/django/misitio/templates',
)
```

Hay algunas cosas que notar:

- Puedes especificar cualquier directorio que quieras, siempre y cuando la cuenta de usuario en la cual se ejecuta el servidor web tengan acceso al directorio y a su contenido. Si no puedes pensar en un lugar apropiado para poner las plantillas, te recomendamos crear un directorio templates dentro del proyecto de Django (esto es, dentro del directorio misitio que creaste en el *capítulo 2* , si vienes siguiendo los ejemplos a lo largo del libro).

- Si tu variable TEMPLATE_DIRS contiene únicamente un directorio, ¡no olvides poner una coma al final de la cadena de texto!

  Mal:

  ```
  # ¡Olvidaste la coma!
  TEMPLATE_DIRS = (
      '/home/django/misitio/templates'
  )
  ```

Bien:

```
# La coma en el lugar correcto.
TEMPLATE_DIRS = (
    '/home/django/misitio/templates',
)
```

Python requiere una coma en las tuplas de un solo elemento para diferenciarlas de una expresión de paréntesis. Este es un error muy común en los usuarios nuevos.

- Si estás en Windows, incluye la letra de tu unidad y usa el estilo de Unix para las barras en vez de barras invertidas, como sigue:

```
TEMPLATE_DIRS = (
    'C:/www/django/templates',
)
```

- Es muy sencillo usar rutas absolutas (esto es, las rutas de directorios comienzan desde la raíz del sistema de archivos). Pero si quieres ser un poco más flexible e independiente, puedes tomar el hecho de que el archivo de configuración de Django es sólo código Python y construir la variable TEMPLATE_DIRS dinámicamente, por ejemplo:

```
import os.path

TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), 'templates').replace('\\','/'),
)
```

Este ejemplo usa la variable "mágica" de Python __file__, la cual es automáticamente asignada al nombre del archivo del módulo de Python en el que se encuentra el código.

Con la variable TEMPLATE_DIRS configurada, el próximo paso es cambiar el código de vista, para que use la funcionalidad automática de carga de plantillas de Django, para no incluir la ruta de la plantilla en la vista, solo el nombre.

Volvamos a la vista fecha_actual y hagamosle algunos cambios:

views.py

```
import datetime

from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse

def fecha_actual(request):
    ahora = datetime.datetime.now()
    t = get_template('fecha_actual.html')
    html = t.render(Context({'fecha_actual': ahora}))
    return HttpResponse(html)
```

En este ejemplo, usamos la función django.template.loader.get_template() en vez de cargar la plantilla desde el sistemas de archivos manualmente. La función get_template() toma el nombre de la plantilla como argumento, se da cuenta en

dónde está la plantilla en el sistema de archivos, la abre, y retorna el objeto Template compilado.

La plantilla de nuestro ejemplo es fecha_actual.html, pero no hay nada especial acerca de la extensión .html. Tu puedes darle la extensión que quieras a tus aplicaciones o puedes omitir las extensiones.

Para determinar la localización de las plantillas en tu sistema de archivos get_template() combina el directorio de plantillas de la variable TEMPLATE_DIRS con el nombre que le pasamos al método get_template(). Por ejemplo si la variable TEMPLATE_DIRS es '/home/django/misitio/templates', el método get_template() buscara las plantillas en /home/django/misitio/templates/fecha_actual.html.

Imagen 4.1 Página de error que muestra cuando una plantilla no se encuentra

Si get_template() no puede encontrar la plantilla con el nombre pasado, esta levanta una excepción TemplateDoesNotExist. Para ver que cómo se ve esto, ejecuta el servidor de desarrollo de Django otra vez, ejecutando python manage.py runserver en el directorio de tu proyecto de Django. Luego, escribe en tu navegador la página que activa la vista fecha_actual (o sea, http://127.0.0.1:8000/fecha/). Asumiendo que tu variable de configuración DEBUG está asignada a True y que todavía no hayas creado la plantilla fecha_actual.html, deberías ver una página de error de Django resaltando el error TemplateDoesNotExist.

Esta página de error es similar a la que explicamos en el *capítulo 3*, con una pieza adicional de información de depuración: una sección “Postmortem del cargador de plantillas”. Esta sección te indica qué plantilla intentó cargar Django acompañado de una razón para cada intento fallido (por ej. “File does not exist”). Esta información es invaluable cuando hacemos depuración de errores de carga de plantillas.

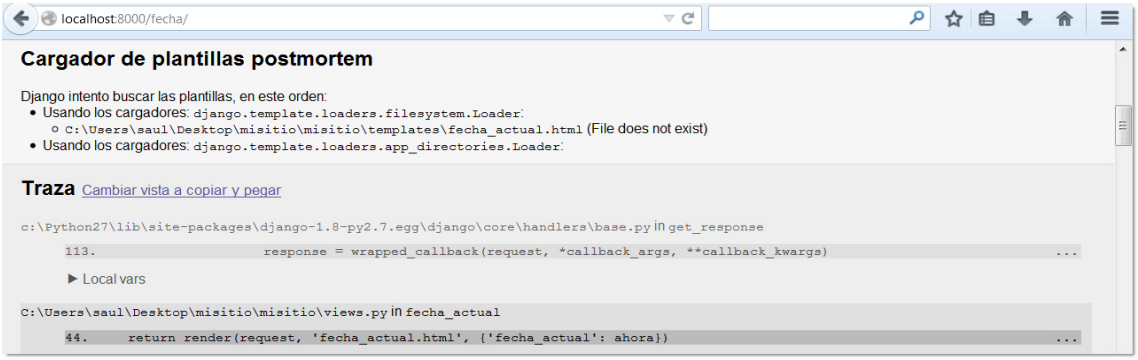


Imagen 4.2 Página de error que muestra la sección postmortem.

Como probablemente puedas distinguir de los mensajes de error de la figura anterior Django intentó buscar una plantilla combinando el directorio de la variable TEMPLATE_DIRS con el nombre de la plantilla pasada a get_template(). Entonces si tu variable TEMPLATE_DIRS contiene '/home/django/templates', Django buscará '/home/django/templates/fecha_actual.html'. Si TEMPLATE_DIRS contiene más

que un directorio, cada uno de estos es examinado hasta que se encuentre la plantilla o hasta que no haya más directorios.

Continuando, crea el archivo fecha_actual.html en tu directorio de plantillas usando el siguiente código:

misitio/misitio/templates/fecha_actual.html

```
<html><body>Hoy es {{ fecha_actual }}.</body></html>
```

Refresca la página en tu navegador web, y deberías ver la página completamente renderizada.

Render()

Hemos visto como cargar una plantilla, rellenar un Context y retornar un objeto HttpResponse con el resultado de la plantilla renderizada. Lo hemos optimizado para usar get_template() en lugar de mezclar la plantilla y hemos usado las rutas de búsqueda de las plantillas. Pero seguimos requiriendo tipear una cantidad considerable de cosas. Sin embargo esto es tan común que Django provee un atajo que te deja hacer todas estas cosas, en una sola línea de código.

Este atajo es la función llamada render(), la cual se encuentra en el módulo django.shortcuts. La mayoría de las veces, usarás render() en vez de cargar las plantillas manualmente y crear los objetos Context y HttpResponse manualmente. – a menos que te paguen por el total de líneas que escribas.

Aquí está el ejemplo que hemos venido usando fecha_actual reescrito utilizando el método-atajo render():

views.py

```
import datetime
from django.shortcuts import render

def fecha_actual(request):
    ahora = datetime.datetime.now()
    return render(request, 'fecha_actual.html', {'fecha_actual': ahora})
```

¡Qué diferencia! Veamos paso a paso, los cambios que hicimos al código:

- Ya no tenemos que importar get_template, Template, Context, o HttpResponse. En vez de eso, solo importamos el atajo django.shortcuts.render, mientras que import datetime se mantiene.

- En la función fecha_actual, seguimos calculando la variable: ahora, pero de la carga de la plantilla, la creación del contexto, la renderización, y la creación de HttpResponse se encarga la llamada a render(). Como render() retorna un objeto HttpResponse, podemos simplemente retornar ese valor en la vista, ¿sencillo no?

El primer argumento de render() debe ser el nombre de la plantilla a utilizar. El segundo argumento, si es pasado, debe ser un diccionario para usar en la creación de un Context para esa plantilla. Si no se le pasa un segundo argumento, render utilizará un diccionario vacío.

Subdirectorios en get_template()

Puede ser un poco inmanejable guardar todas las plantillas en un solo directorio. Quizás quieras guardar las plantillas en subdirectorios del directorio de tus plantillas, y esto está bien. De hecho, recomendamos hacerlo; algunas de las características más avanzadas de Django (como las vistas genéricas del sistema, las cuales veremos en el capítulo 11) esperan esta distribución de las plantillas como una convención por omisión.

Guardar las plantillas en subdirectorios de tu directorio de plantilla es fácil. En tus llamadas a get_template(), sólo incluye el nombre del subdirectorio y una barra antes del nombre de la plantilla, así:

```
t = get_template('aplicacion/fecha_actual.html')
```

Debido a que render() es un pequeño contenedor de get_template(), puedes hacer lo mismo con el primer argumento de render.

```
return render(request, 'aplicacion/fecha_actual.html', {'fecha_actual': ahora})
```

No hay límites para la profundidad del árbol de subdirectorios. Siéntete libre de usar tantos como quieras o necesites.

NOTA: Para usuarios de Windows, es necesario asegurar el uso de barras comunes en vez de barras invertidas. get_template() asume el estilo de designación de archivos usado en Unix.

La etiqueta de plantilla include

Ahora que hemos visto en funcionamiento el mecanismo para cargar plantillas, podemos introducir un tipo de plantilla incorporada que tiene una ventaja para esto: {% include %}. Esta etiqueta te permite incluir el contenido de otra plantilla. El argumento para esta etiqueta debería ser el nombre de la plantilla a incluir, y el nombre de la plantilla puede ser una variable string hard-coded (entre comillas), entre simples o dobles comillas. En cualquier momento que tengas el mismo código en varias etiquetas, considera utilizar la etiqueta {% include %} para eliminar la redundancia entre las plantillas.

Estos dos ejemplos incluyen el contenido de la plantilla nav.html. Los ejemplos son equivalentes e ilustran que cualquier modo de comillas está permitido:

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

Este ejemplo incluye el contenido de la plantilla includes/nav.html:

```
{% include 'includes/nav.html' %}
```

Este ejemplo incluye el contenido de la plantilla cuyo nombre se encuentra en la variable template_name:

```
{% include template_name %}
```

Como en get_template(), el nombre del archivo de la plantilla es determinado agregando el directorio de plantillas tomado de TEMPLATE_DIRS para el nombre de plantilla solicitado.

Las plantillas incluidas son evaluadas con el contexto de la plantilla en la cual está incluida.

Considera estos dos ejemplos:

mipagina.html

```
<html>
<body>
 {% include "includes/nav.html" %}
   <h1>{{ titulo }}</h1>
</body>
</html>
```

includes/nav.html

```
<div id="nav">
   Tu estas en: {{ seccion_actual }}
</div>
```

Si renderizas mipagina.html con un contexto que contiene la variable sección_ actual, la variable estará disponible en la plantilla "incluida" tal como esperarías.

Si una plantilla no encuentra la etiqueta {% include %}, Django hará una de estas dos cosas:

1. Si DEBUG es True, verás la excepción TemplateDoesNotExist sobre la página de error de Django.

2. Si DEBUG es False, la etiqueta fallará silenciosamente, sin mostrar nada en el lugar de la etiqueta.

Herencia de plantillas

Nuestras plantillas de ejemplo hasta el momento han sido fragmentos de HTML, pero en el mundo real, usarás el sistema de plantillas de Django para crear páginas HTML enteras. Esto conduce a un problema común en el desarrollo web: ¿Cómo reducimos la duplicación y la redundancia de las áreas comunes de las páginas, como por ejemplo, los paneles de navegación?

Una forma clásica de solucionar este problema es usar *includes*, insertando dentro de las páginas HTML a "incluir" una página dentro de otra. Es más, Django admite esta aproximación, con la etiqueta {% include %} anteriormente descrita. Pero la mejor forma de solucionar este problema con Django es usar una estrategia más elegante llamada *herencia de plantillas*.

En esencia, la herencia de plantillas te deja construir una plantilla base "esqueleto" que contenga todas las partes comunes de tu sitio y definir "bloques" que las plantillas hijas puedan sobrescribir.

Veamos un ejemplo de esto creando una plantilla completa para nuestra vista fecha_actual, edita el archivo fecha_actual.html así:

misitio/misitio/templates/fecha_actual.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
   <title>Fecha Actual</title>
</head>
<body>
   <h1>Mi util sitio</h1>
   <p>Hoy es: {{ fecha_actual }}.</p>
<hr>
   <p>Gracias por visitar nuestro sitio web.</p>
</body>
</html>
```

Esto se ve bien, pero ¿Qué sucede cuando queremos crear una plantilla para otra vista –digamos, ¿La vista horas_adelante del *capítulo 3*? Si queremos hacer nuevamente una plantilla completa agradable y válida, crearíamos algo como esto:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
   <title>Fecha Futura</title>
</head>
<body>
   <h1>Mi util sitio</h1>
   <p>En {{ horas_adelante }} hora(s), será {{ hora_siguiente }}.</p>
   <hr>
   <p>Gracias por visitar nuestro sitio web.</p>
</body>
</html>
```

Claramente, estaríamos duplicando una cantidad considerable de código HTML. Imagina si tuvieramos mas cosas típicas, como barras de navegación, algunas hojas de estilo, quizás algo de JavaScript – terminaríamos poniendo todo tipo de HTML redundante en cada plantilla–.

La solución a este problema, es usar “includes” en el servidor para sacar factor común de las plantillas y guardarlas en recortes de plantillas separados, que luego son incluidos en cada plantilla. Quizás quieras guardar la parte superior de la plantilla en un archivo llamado cabecera_pagina.html:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
   <html lang="en">
<head>
```

Y quizás quieras guardar la parte inferior en un archivo llamado pie_pagina.html:

```
   <hr>
   <p>Gracias por visitar nuestro sitio web.</p>
</body>
</html>
```

Con una estrategia basada en “includes”, la cabecera y la parte de abajo son fáciles. Es el medio el que queda desordenado. En este ejemplo, ambas páginas contienen un título – <h1>Mi útil sitio</h1> pero ese título no puede encajar dentro de

header.html porque el <title> en las dos páginas es diferente. Si incluimos <h1> en la cabecera, tendríamos que incluir <title>, lo cual no permitiría personalizar este en cada página. ¿Ves a dónde queremos llegar?

El sistema de herencia de Django soluciona estos problemas. Lo puedes pensar a esto como la versión contraria a la del lado del servidor. En vez de definir los pedazos que son *comunes*, solo defines los pedazos que son *diferentes*.

El primer paso es definir una *plantilla base* – un "esqueleto" de tu página que las *plantillas hijas* llenaran luego.

Aquí hay una plantilla para nuestro ejemplo actual:

misitio/misitio/templates/base.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
   <title>{% block title %}{% endblock %}</title>
</head>
<body>
   <h1>Mi sitio Web</h1>
   {% block content %}{% endblock %}
   {% block footer %}
   <hr>
   <p>Gracias por visitar nuestro sitio web.</p>
{% endblock %}
</body>
</html>
```

Esta plantilla, que llamamos base.html, define un esqueleto HTML o mejor dicho define la estructura del documento, que usaremos para todas las páginas del sitio. Es trabajo de las plantillas hijas sobrescribir, agregar, dejar vacío el contenido de los bloques. (Si continuas siguiendo los ejemplos, guarda este archivo en tu directorio de plantillas).

Usamos una etiqueta de plantillas nueva aquí: la etiqueta {% block %}. Todas las etiquetas {% block %}, le indican al motor de plantillas que una plantilla hija, quizás sobrescriba esa parte de la plantilla.

Ahora que tenemos una plantilla base, podemos modificar nuestra plantilla existente fecha_actual para usar la etiqueta extends y usar la plantilla base.html:

misitio/misitio/templates/fecha_actual.html

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

{% block title %}La fecha actual{% endblock %}

{% block content %}
   <p>Hoy es: {{ fecha_actual }}</p>
{% endblock %}
```

Siguiendo con el tema de plantillas, vamos a crear una plantilla para la vista horas_adelante del *capítulo 3*. (Si estás siguiendo los ejemplos, cambia el código de la vista horas_adelante para que use el sistema de plantillas).

La función vista del el archivo views.py queda de la siguiente forma, incluyendo la vista anterior:

views.py

```
import datetime
from django.shortcuts import render

def fecha_actual(request):
    ahora = datetime.datetime.now()
    return render(request, 'fecha_actual.html', {'fecha_actual': ahora})

def horas_adelante(request, horas):
    try:
        horas = int(horas)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=horas)
    return render(request, 'horas_adelante.html', {'hora_siguiente': dt, 'horas': horas })
```

Y esta es la plantilla para horas_adelante.html:

misitio/misitio/templates/horas_adelante.html

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

{% block title %}Fecha Futura{% endblock %}

{% block content %}
    <p>En {{ horas }} horas(s), la fecha sera: {{ hora_siguiente }}.</p>
{% endblock %}
```

¿No es hermoso? Cada plantilla contiene sólo el código que es *diferente* para esa plantilla. No necesita redundancia. Si necesitas hacer un cambio radical en el diseño del sitio Web, sólo cambia la plantilla base.html, y todas las demás plantillas reflejarán los cambios inmediatamente.

Veamos cómo trabaja:

Cuando cargamos una plantilla, por ejemplo fecha_actual.html, el motor de plantillas ve la etiqueta {% extends %}, nota que esta plantilla es la hija de otra. El motor inmediatamente carga la plantilla padre –en este caso, base.html.

Hasta este punto, el motor de la plantilla nota las tres etiquetas {% block %} en base.html y reemplaza estos bloques por el contenido de la plantilla hija. Entonces, el título que definimos en el bloque {% block title %} será usado, así como el que definimos en el bloque {% block content %}.

Nota que la plantilla hija no define el bloque footer, entonces el sistema de plantillas usa el valor de la plantilla padre por defecto. El contenido de la etiqueta {% block %} en la plantilla padre es usado siempre que no se sobrescribe en una plantilla hija.

La herencia no afecta el funcionamiento del contexto, y puedes usar tantos niveles de herencia como necesites. Una forma común de utilizar la herencia es el siguiente enfoque de tres niveles:

1. Crea una plantilla base.html que contenga el aspecto principal de tu sitio. Esto es lo que rara vez cambiará, si es que alguna vez cambia.

2. Crea una plantilla base_SECTION.html para cada "sección" de tu sitio (por ej. base_fotos.html y base_foro.html). Esas plantillas heredan de base.html e incluyen secciones específicas de estilo y diseño.

3. Crea una plantilla individual para cada tipo de página, tales como páginas de formulario o galería de fotos. Estas plantillas heredan de la plantilla solo la sección apropiada.

Esta aproximación maximiza la reutilización de código y hace más fácil agregar elementos para compartir distintas áreas, como puede ser un navegador de sección, un contenido o una cabecera.

Aquí hay algunos consejos para trabajar con la herencia de plantillas:

- Si usas etiquetas {% extends %} en la plantilla, esta debe ser la primera etiqueta de esa plantilla. En otro caso, la herencia no funcionará.

- Generalmente, cuanto más etiquetas {% block %} tengas en tus plantillas, mejor. Recuerda, las plantillas hijas no tienen que definir todos los bloques del padre, entonces puedes rellenar un número razonable de bloques por omisión, y luego definir sólo lo que necesiten las plantillas hijas. Es mejor tener más conexiones que menos.

- Si encuentras código duplicado en un número de plantillas, esto probablemente signifique que debes mover ese código a un {% block %} en la plantilla padre.

- Si necesitas obtener el contenido de un bloque desde la plantilla padre, la variable {{ block.super }} hará este truco. Esto es útil si quieres agregar contenido del bloque padre en vez de sobreescribirlo completamente.

- No puedes definir múltiples etiquetas {% block %} con el mismo nombre en la misma plantilla. Esta limitación existe porque una etiqueta bloque trabaja en ambas direcciones. Esto es, una etiqueta bloque no sólo provee un agujero a llenar, sino que también define el contenido que llenará ese agujero en el *padre*. Si hay dos nombres similares de etiquetas {% block %} en una plantilla, el padre de esta plantilla puede no saber cuál de los bloques usar (aunque usara el primero que encuentre).

- El nombre de plantilla pasado a {% extends %} es cargado usando el mismo método que get_template(). Esto es, el nombre de la plantilla es agregado a la variable TEMPLATE_DIRS.
- En la mayoría de los casos, el argumento para {% extends %} será un string o cadena, pero también puede ser una variable, si no sabes el nombre de la plantilla padre hasta la ejecución. Esto te permite hacer cosas divertidas y dinámicas.

¿Qué sigue?

Los sitios Web modernos, son *manejados con una base de datos*: el contenido de la página Web está guardado en una base de datos relacional. Esto permite una clara separación entre los datos y la lógica de los datos (de la misma forma en que las vistas y las plantillas permiten una separación de la lógica y la vista). Él *próximo capítulo* cubre las herramientas que Django brinda para interactuar con bases de datos.

CAPÍTULO 5

Interactuando con una base de datos: Modelos

El capítulo 3, cubrió los conceptos fundamentales sobre la construcción dinámica de sitios web con Django, así como la configuración de vistas y URLconfs. Como explicamos, una vista es responsable de implementar *alguna lógica arbitraria* y luego retornar una respuesta. En el ejemplo, nuestra lógica arbitraria era calcular la fecha y la hora actual.

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

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

Django es apropiado para crear sitios web que manejen una base de datos, ya que incluye una manera fácil pero poderosa de realizar consultas a bases de datos utilizando Python. Este capítulo explica esta funcionalidad: **la capa de la base de datos de Django**.

Nota: Aunque no es estrictamente necesario conocer teoría básica sobre bases de datos y SQL para usar la capa de la base de datos de Django, es altamente recomendado. Una introducción a estos conceptos va más allá del alcance de este libro, pero continúa leyendo si eres nuevo en el tema.

De seguro serás capaz de seguir adelante y captar los conceptos básicos en base al contexto y los ejemplos.

La manera "tonta" de hacer una consulta a la base de datos en las vistas

Así como en el *capítulo 3* detallamos la manera "tonta" de producir una salida con la vista (codificando *en duro* el texto directamente dentro de la vista), hay una manera "tonta" de recuperar datos desde una base de datos en una vista. Esto es simple: sólo

usa una biblioteca de Python existente para ejecutar una consulta SQL y haz algo con los resultados.

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

```
import MySQLdb
from django.shortcuts import render

def lista_biblioteca(request):
    db = MySQLdb.connect(user='yo', db='datos.db', passwd='admin',   host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT nombre FROM biblioteca ORDER BY nombre')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render(request, 'lista_libros.html', {'nombres': nombres})
```

Este enfoque funciona, pero inmediatamente se hacen evidentes algunos problemas:

- Estamos *codificando en duro* (*hard-coding*) los parámetros de la conexión a la base de datos. Lo ideal sería que esos parámetros se guardarse en la configuración de Django.

- Tenemos que escribir una gran cantidad de código repetitivo: crear una conexión, un cursor, ejecutar una sentencia, y cerrar la conexión. Lo ideal sería que todo lo que tuviéramos que hacer fuera especificar los resultados que queremos obtener.

- Nos ata a MySQL. Si, en el camino, cambiamos de MySQL a PostgreSQL, tenemos que usar un adaptador de base de datos diferente (por ej. psycopg en vez de MySQLdb), alterar los parámetros de conexión y – dependiendo de la naturaleza de las sentencia de SQL, posiblemente reescribir el SQL. La idea es que el servidor de base de datos que usemos esté abstraído, entonces el pasarnos a otro servidor podría significar realizar un cambio en un único lugar.

Como esperabas, la capa de la base de datos de Django apunta a resolver estos problemas. Este es un adelanto de cómo la vista anterior puede ser reescrita usando la API de Django:

```
from django.shortcuts import render_to_response
from misitio.biblioteca.models import Libro

def lista_biblioteca(request):
    lista_libros = Libro.objects.order_by('nombre')
    return render_to_response('lista_libros.html', {'lista_libros': lista_libros})
```

Explicaremos este código un poco más adelante, en este capítulo. Por ahora, solo observa la forma en que lo escribimos, usando la capa de modelos de Django.

El patrón de diseño MTV

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

Como mencionamos en los capítulos anteriores, Django fue diseñado para promover el acoplamiento débil y la estricta separación entre las piezas de una aplicación. Si sigues esta filosofía, es fácil hacer cambios en un lugar particular de la aplicación sin afectar otras piezas.

En las funciones de vista, por ejemplo, discutimos la importancia de separar la lógica de negocios, de la lógica de presentación usando un sistema de plantillas. Con la capa de la base de datos, aplicamos esa misma filosofía para el acceso lógico a los datos.

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

¿POR QUÉ EL ACRÓNIMO?

El objetivo de definir en forma explícita patrones tales como MVC es principalmente, para simplificar la comunicación entre los desarrolladores. En lugar de tener que decirle a tus compañeros de trabajo, "Vamos a hacer una abstracción del acceso a la base de datos, luego vamos a crear una capa que se encarga de mostrar los datos, y vamos a poner una capa en el medio para que regule esto", puedes sacar provecho de un vocabulario compartido y decir, "Vamos a usar un patrón de diseño MVC aquí".

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

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

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

- *M* significa "Model" (Modelo), la capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.

- *T* significa "Template" (Plantilla), la capa de presentación. Esta capa contiene las decisiones relacionadas a la presentación: como algunas cosas son mostradas sobre una página web o otro tipo de documento.

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

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

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

Configuración de la base de datos

Con toda esta filosofía en mente, vamos a comenzar a explorar la capa de la base de datos de Django. Primero, necesitamos tener en cuenta algunas configuraciones iníciales: necesitamos indicarle a Django qué servidor de base de datos usar y cómo conectarse al mismo.

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

Como con TEMPLATE_DIRS en los capítulos anteriores, la configuración de la base de datos se encuentra en el archivo de configuración de Django, llamado, por omisión, settings.py. Edita este archivo y busca las opciones de la variable DATABASES, el cual es un diccionario que contiene los ajustes necesarios, para configurar la base datos:

```
ENGINE = ''
NAME = ''
USER = ''
PASSWORD = ''
HOST = ''
DATABASE_PORT = ''
```

Aquí hay un resumen de cada propiedad.

- ENGINE: le indica a Django qué base de datos utilizar. Si usas una base de datos con Django, ENGINE debe configurarse con una cadena de los mostrados en la Tabla 5-1.

Configuración	Base de datos	Adaptador requerido
django.db.backends.postgresql_psycopg2	PostgreSQL	Psycopg version 2.x, http://www.djangoproject.com/r/python-pgsql/.
django.db.backends.mysql	MySQL	MySQLdb, http://www.djangoproject.com/r/python-mysql/.
django.db.backends.sqlite3	SQLite	No necesita adaptador
django.db.backends.oracle	Oracle	cx_Oracle, http://www.djangoproject.com/r/python-oracle/.

Tabla 5.1 Adaptadores requeridos en Django

> **Nota**: Cualquiera que sea la base de datos que uses, necesitarás descargar e instalar el adaptador apropiado. Cada uno de estos está disponible libremente en la web; sólo sigue el enlace en la columna "Adaptador requerido" en la Tabla 5-1.

- **NAME** la indica a Django el nombre de tu base de datos. Si estás usando SQLite, especifica la ruta completo del sistema de archivos hacia el archivo de la base de datos (por ej. '/home/django/datos.db').

- **USER** le indica a Django cual es el nombre de usuario a usar cuando se conecte con tu base de datos. Si estás usando SQLite, deja este en blanco.

- **PASSWORD** le indica a Django cual es la contraseña a utilizar cuando se conecte con tu base de datos. Si estás utilizando SQLite o tienes una contraseña vacía, deja este en blanco.

- **HOST** le indica a Django cual es el host a usar cuando se conecta a tu base de datos. Si tu base de datos está sobre la misma computadora que la instalación de Django (o sea localhost), deja este en blanco. Si estás usando SQLite, deja este en blanco.

  MySQL es un caso especial aquí. Si este valor comienza con una barra ('/') y estás usando MySQL, MySQL se conectará al socket especificado por medio de un socket Unix, por ejemplo:

  ```
  DATABASE_HOST = '/var/run/mysql'
  ```

  Si estás utilizando MySQL y este valor *no* comienza con una barra, entonces este valor es asumido como el host.

- **PORT** le indica a Django qué puerto usar cuando se conecte a la base de datos. Si estás utilizando SQLite, deja este en blanco. En otro caso, si dejas este en blanco, el adaptador de base de datos subyacente usará el puerto por omisión acorde al servidor de base de datos. En la mayoría de los casos, el puerto por omisión está bien, por lo tanto puedes dejar este en blanco.

La variable DATABASES, por omisión usa la configuración más simple posible, la cual está configurada para utilizar SQLite, por lo que no tendrás que configurar nada, si vas a usar SQLite como base de datos:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Sin embargo si quieres usar otra base de datos como MySQL, Oracle, o PostgreSQL es necesario especificar algunos parámetros adicionales, que serán requeridos en el archivo de configuración.

El siguiente ejemplo asume que quieres utilizar PostgreSQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'basedatos.bd',
        'USER': 'yo',
        'PASSWORD': 'admin',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

Como podrás darte cuenta, lo único que necesitas cambiar es la 'ENGINE', si quieres usar MySQL e introducir los datos apropiados de acuerdo a la base de datos que estés usando.

Una vez que hayas ingresado estas configuraciones, compruébalas. Primero, desde el directorio del proyecto que creaste en el *capítulo 2*, ejecuta el comando:

```
python manage.py shell
```

Notarás que comienza un intérprete interactivo de Python. Las apariencias pueden engañar. Hay una diferencia importante entre ejecutar el comando manage.py shell dentro del directorio del proyecto de Django y el intérprete genérico python. El último es el Python shell básico, pero el anterior le indica a Django cuales archivos de configuración usar antes de comenzar el shell.

Este es un requerimiento clave para realizar consultas a la base de datos: Django necesita saber cuáles son los archivos de configuraciones a usar para obtener la información de la conexión a la base de datos.

Detrás de escena, manage.py shell simplemente asume que tu archivo de configuración está en el mismo directorio que manage.py. Hay otras maneras de indicarle a Django qué módulo de configuración usar, pero este tópico lo cubriremos más adelante. Por ahora, usa manage.py shell cuando necesites hacer modificaciones y consultas específicas a Django.

Una vez que hayas entrado al shell, escribe estos comandos para probar la configuración de tu base de datos:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
>>>
```

Si no sucede nada, entonces tu base de datos está configurada correctamente. De lo contrario revisa el mensaje de error para obtener un indicio sobre qué es lo que está mal. La Tabla 5-2 muestra algunos mensajes de error comunes.

Mensaje de error	Solución
You haven't set the ENGINE setting yet.	Configura la variable ENGINE con otra cosa que no sea un string vacío. Observa los valores de la tabla 5-1.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Ejecuta el comando python manage.py shell en vez de python.
Error loading _____ module: No module named _____.	No tienes instalado el módulo apropiado para la base de datos especificada (por ej. psycopg o MySQLdb).(Es tu responsabilidad instalarlos)
_____ isn't an available database backend.	Configura la variable ENGINE con un motor válido descrito previamente. ¿Habrás cometido un error de tipeo?
database _____ does not exist	Cambia la variable NAME para que *apunte* a una base de datos existente, o ejecuta la sentencia CREATE DATABASE apropiada para crearla.
role _____ does not exist	Cambia la variable USER para que *apunte* a un usuario que exista, o crea el usuario en tu base de datos.
could not connect to server	Asegúrate de que HOST y PORT esta configurados correctamente y que el servidor esté corriendo.

Tabla 5.2 Mensajes de error

Tu primera aplicación

Ahora que verificamos que la conexión está funcionando, es hora de crear una **Aplicación de Django** – una colección de archivos de código fuente, incluyendo modelos y vistas, que conviven en un solo paquete de Python y representen una aplicación completa de Django.

Vale la pena explicar la terminología aquí, porque esto es algo que suele hacer tropezar a los principiantes. Ya hemos creado un *proyecto*, en el *capítulo 2*, entonces, ¿cuál es la diferencia entre un *proyecto* y una *aplicación*? Bueno, la diferencia es la que existe entre la configuración y el código:

- **Un proyecto** es una instancia de un cierto conjunto de aplicaciones de Django, más las configuraciones de esas aplicaciones. Técnicamente, el único requerimiento de un proyecto es que este suministre un archivo de configuración o settings.py, el cual define la información hacia la conexión a la base de datos, la

lista de las aplicaciones instaladas, la variable TEMPLATE_DIRS, y así sucesivamente.

- **Una aplicación** es un conjunto portable de alguna funcionalidad de Django, típicamente incluye modelos y vistas, que conviven en un solo paquete de Python (Aunque el único requerimiento es que contenga una archivo models.py).

Por ejemplo, Django incluye un número de aplicaciones, tales como un framework geográfico "Geodejango" y una interfaz de administración automática. Una cosa clave para notar sobre las aplicaciones es que son portables y reusables en múltiples proyectos.

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

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

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

Siguiendo con el ejemplo, dentro del directorio del proyecto misitio que creaste en el *capítulo 2*, escribe este comando para crear una nueva aplicación a la que llamaremos **biblioteca:**

```
python manage.py startapp biblioteca
```

Este comando no produce ninguna salida, pero crea un directorio llamado biblioteca dentro del directorio misitio y dentro de este crea otro directorio más, llamado migrations.

Echemos un vistazo al contenido:

```
biblioteca/
    __init__.py
    admin.py
    models.py
    tests.py
    views.py
    migrations/
        __init__.py
```

Estos archivos contendrán los modelos, las vistas, las pruebas, y las migraciones para esta aplicación, aprenderemos como usarlos en los siguientes capítulos.

Echa un vistazo a **models.py**, **admin.py**, **views.py**, **tests.py** en tu editor de texto favorito. Estos archivos están vacíos, excepto por las importaciones. Este es el espacio disponible para ser creativo con tu aplicación de Django.

Definir modelos en Python

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

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

- La introspección requiere *overhead* y es imperfecta. Con el objetivo de proveer una API conveniente de acceso a los datos, Django necesita conocer *de alguna forma* la capa de la base de datos, y hay dos formas de lograr esto. La primera sería describir explícitamente los datos en Python, y la segunda sería la introspección de la base de datos en tiempo de ejecución para determinar el modelo de la base de datos.

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

- Escribir Python es divertido, y dejar todo en Python limita el número de veces que tu cerebro tiene que realizar un "cambio de contexto". Si te mantienes en un solo entorno/mentalidad de programación tanto tiempo como sea posible, ayuda para la productividad. Teniendo que escribir SQL, luego Python, y luego SQL otra vez es perjudicial.

- Tener modelos de datos guardados como código en vez de en tu base de datos hace fácil dejar tus modelos bajo un control de versiones. De esta forma, puedes fácilmente dejar rastro de los cambios a tu capa de modelos.

- SQL permite sólo un cierto nivel de metadatos acerca de un *layout* de datos. La mayoría de sistemas de base de datos, por ejemplo, no provee un tipo de datos especializado para representar una dirección web o de email. Los modelos de Django sí. La ventaja de un tipo de datos de alto nivel es la alta productividad y la reusabilidad de código.

- SQL es inconsistente a través de distintas plataformas. Si estás redistribuyendo una aplicación web, por ejemplo, es mucho más pragmático

distribuir un módulo de Python que describa tu capa de datos que separar conjuntos de sentencias CREATE TABLE para MySQL, PostgreSQL y SQLite.

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

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

Tu primer Modelo

Para empezar a trabajar con ejemplos, en este capítulo y en el siguiente nos enfocaremos en crear una configuración de datos básica sobre libro/autor/editor (una biblioteca). Usaremos este ejemplo porque las relaciones conceptuales entre libros, autores y editores son bien conocidas, y es una configuración de base datos comúnmente utilizada, en una biblioteca online, además de que se usa en muchos lugares como texto introductorio a SQL. Por otra parte, ¡estás leyendo un libro que fue escrito por autores y producido por un editor!

Asumiremos los siguientes conceptos, campos y relaciones:

- Un autor tiene un nombre, apellidos, un correo electrónico...
- Un editor tiene un nombre, un domicilio, una ciudad, un estado o provincia, un país y un sitio Web.
- Un libro tiene un título y una fecha de publicación. También tiene uno o más autores (una relación muchos-a-muchos con autores) y un único editor (una relación uno a muchos – también conocida como clave foránea – con editores).

El primer paso para utilizar esta configuración de base de datos con Django es expresarla como código Python. En el archivo models.py que se creó con el comando startapp, ingresa lo siguiente:

biblioteca/models.py

```
from django.db import models

class Editor(models.Model):
    nombre = models.CharField(max_length=30)
    domicilio = models.CharField(max_length=50)
    ciudad = models.CharField(max_length=60)
    estado = models.CharField(max_length=30)
    pais = models.CharField(max_length=50)
    website = models.URLField()

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField()
```

```
class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editor = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField()
    portada = models.ImageField(upload_to='portadas')
```

Examinemos rápidamente este código para conocer lo básico. La primera cosa a notar es que cada modelo es representado por una clase Python que es una subclase de django.db.models.Model. La clase antecesora, Model, contiene toda la maquinaria necesaria para hacer que estos objetos sean capaces de interactuar con la base de datos y que hace que nuestros modelos sólo sean responsables de definir sus campos, en una sintaxis compacta y agradable. Lo creas o no, éste es todo el código que necesitamos para tener acceso básico a los datos con Django.

Cada modelo generalmente corresponde a una tabla única de la base de datos, y cada atributo de un modelo generalmente corresponde a una columna en esa tabla. El nombre de atributo corresponde al nombre de columna, y el tipo de campo (ej.: CharField) corresponde al tipo de columna de la base de datos (ej.: varchar). Por ejemplo, el modelo Editor es equivalente a la siguiente tabla (asumiendo la sintaxis de PostgreSQL para CREATE TABLE):

```
CREATE TABLE "Editor" (
    "id" serial NOT NULL PRIMARY KEY,
    "nombre" varchar(30) NOT NULL,
    "domicilio" varchar(50) NOT NULL,
    "ciudad" varchar(60) NOT NULL,
    "estado" varchar(30) NOT NULL,
    "pais" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

En efecto, Django puede generar esta sentencia CREATE TABLE automáticamente como veremos en un momento.

La excepción a la regla una-clase-por-tabla es el caso de las relaciones muchos-a-muchos. En nuestros modelos de ejemplo, libro tiene un ManyToManyField llamado autor. Esto significa que un libro tiene uno o más autores, pero la tabla de la base de datos libro no tiene una columna autores. En su lugar, Django crea una tabla adicional – una "tabla de join" muchos-a-muchos – que maneja la correlación entre biblioteca y autores.

Para una lista completa de todos los tipos de campo y las distintas opciones de sintaxis de modelos, consulta el *apéndice B*.

Finalmente, debes notar que no hemos definido explícitamente una clave primaria en ninguno de estos modelos. A no ser que le indiques lo contrario, Django dará automáticamente a cada modelo un campo de clave primaria, llamado id. Es un requerimiento el que cada modelo Django tenga una clave primaria de columna simple.

Instalando el modelo

Ya escribimos el código; ahora necesitamos crear las tablas en la base de datos. Para ello, el primer paso es **activar** estos modelos en nuestro proyecto Django. Hacemos esto agregando la aplicación biblioteca a la lista de aplicaciones instaladas en el archivo de configuración.

Edita el archivo settings.py y examina la variable de configuración INSTALLED_APPS, INSTALLED_APPS le indica a Django qué aplicaciones están activadas para un proyecto determinado. Por defecto esta se ve así:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

Agrega tu aplicación 'biblioteca' a la lista de INSTALLED_APPS, de manera que la configuración termine viéndose así:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'biblioteca',
)
```

(Como aquí estamos tratando con una tupla de un solo elemento, no olvides la coma al final. De paso, los autores de este libro prefieren poner una coma después de *cada* elemento de una tupla, aunque la tupla tenga sólo un elemento. Esto evita el problema de olvidar comas, y no hay penalización por el use de esa coma extra)

Ten en cuenta que **biblioteca** se refiere a la aplicación biblioteca en la que estamos trabajando. Cada aplicación en INSTALLED_APPS es representada por su ruta Python completa – esto es, la ruta de paquetes, separados por puntos, que lleva al paquete de la aplicación.

■**Nota:** Si estás usando PostgreSQL, Oracle o MySQL, debes asegurarte de crear una base de datos en este punto. Lo puedes hacer con el comando "CREATE DATABASE nombre_base_de_datos;" mediante el intérprete interactivo de la base de datos.

Asegúrate de instalar la librería de imágenes **Pillow**, para validar imágenes ya que Django la utiliza para comprobar que los objetos que sean subidos a un campo ImageField sean imágenes validas, de lo contrario Django se quejara si intentas usar un campo ImageField sin tener instalada la librería Pillow. Para instalarla usa el comando:

```
pip install pillow
```

También agrega la ruta al directorio en donde se guardaran las imágenes, especificándolo en el archivo de configuraciones setings.py y usando la variable MEDIA_ROOT:

```
MEDIA_ROOT = 'media/'
```

Y La URL que se encargara de servir dichas imágenes MEDIA_URL, por ejemplo asumiendo que estas usando el servidor de desarrollo:

```
MEDIA_URL = 'http://localhost:8000/media/'
```

Si estas usando SQLite no necesitaras crear nada de antemano –la base de datos se creará automáticamente cuando esta se necesite. Ahora que la aplicación Django ha sido activada en el archivo de configuración, podemos crear las tablas en nuestra base de datos. Primero, validemos los modelos ejecutando este comando:

El comando validate verifica si la sintaxis y la lógica de tus modelos son correctas. Si todo está bien, verás el mensaje 0 errors found. Si no, asegúrate de haber escrito el código del modelo correctamente. La salida del error debe brindarte información útil acerca de qué es lo que está mal en el código.

Cada vez que piensas que tienes problemas con tus modelos, ejecuta manage.py validate. Tiende a capturar todos los problemas comunes del modelo.

Si tus modelos son válidos, ejecuta el siguiente comando para que Django compruebe la sintaxis de tus modelos en la aplicación biblioteca:

```
python manage.py check biblioteca
```

El comando check verifica que todo esté en orden respecto a tus modelos, no crea ni toca de ninguna forma tu base de datos – sólo imprime una salida en la pantalla en la que identifica posibles errores en tus modelos.

Una vez que todo está en orden, necesitamos guardar las migraciones para los modelos en un archivo de control, para que Django pueda encontrarlas al sincronizar el esquema de la base de datos. Ejecuta el comando makemigrations de esta manera:

```
python manage.py makemigrations
```

Verás algo como esto:

```
Migrations for 'biblioteca':
  0001_initial.py:
    - Create model Editor
    - Create model Autor
    - Create model Libro
```

Una vez que usamos el comando makemigrations, para crear las "migraciones", podemos usar el comando sqlmigrate para ver el SQL generado. El comando sqlmigrate toma los nombres de las migraciones y las retorna en un lenguaje SQL, por cada aplicación especificada, de la siguiente forma:

```
python manage.py sqlmigrate biblioteca 0001
```

En este comando, biblioteca es el nombre de la aplicación y 0001, es el número que Django asigna como nombre a cada migración o cambio hecho al esquema de la base de datos (mira dentro de la carpeta migrations). Cuando ejecutes el comando, debes ver algo como esto (formateado para legibilidad, usando SQLite):

```
BEGIN;
CREATE TABLE "biblioteca_editor" (
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
```

```
    "nombre" varchar(30) NOT NULL,
    "domicilio" varchar(50) NOT NULL,
    "ciudad" varchar(60) NOT NULL,
    "estado" varchar(30) NOT NULL,
    "pais" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
)
;
CREATE TABLE "biblioteca_autor" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "nombre" varchar(30) NOT NULL,
    "apellidos" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
)
;
CREATE TABLE "biblioteca_libro_autores" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "libro_id" integer NOT NULL,
    "autor_id" integer NOT NULL REFERENCES "biblioteca_autor" ("id"),
UNIQUE ("libro_id", "autor_id")
)
;
CREATE TABLE "biblioteca_libro" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "titulo" varchar(100) NOT NULL,
    "editor_id" integer NOT NULL REFERENCES "biblioteca_editor" ("id"),
    "fecha_publicacion" date NOT NULL,
    "portada" varchar(100) NOT NULL
)
;
CREATE INDEX "biblioteca_libro_autores_dd67b109" ON "biblioteca_libro_autores"
("libro_id");
CREATE INDEX "biblioteca_libro_autores_40e8bcf3" ON "biblioteca_libro_autores"
("autor_id");

CREATE INDEX "biblioteca_libro_c2be667f" ON "biblioteca_libro" ("editor_id");
COMMIT;
```

Observa lo siguiente:

- Los nombres de tabla se generan automáticamente combinando el nombre de la aplicación (biblioteca) y el nombre en minúsculas del modelo (Editor, Libro, y Autor). Puedes sobrescribir este comportamiento, como se detalla en el Apéndice B.

- Como mencionamos antes, Django agrega una clave primaria para cada tabla automáticamente – los campos id. También puedes sobrescribir esto.

- Por convención, Django agrega "_id" al nombre de campo de las claves foráneas. Como ya puedes imaginar, también puedes sobrescribir esto.

- La relación de clave foránea se hace explícita con una sentencia REFERENCES

- Estas sentencias CREATE TABLE son adaptadas a medida de la base de datos que estás usando, de manera que Django maneja automáticamente los tipos de campo específicos de cada base de datos, como auto_increment (MySQL), serial (PostgreSQL), o integer primary key (SQLite), por ti. Lo mismo sucede con el uso de las comillas simples o dobles en los nombres de columna. La salida del ejemplo está en la sintaxis de PostgreSQL.

El comando sqlmigrate no crea ni toca de ninguna forma tu base de datos – sólo imprime una salida en la pantalla para que puedas ver qué SQL ejecutaría Django si le pidieras que lo hiciera. Si quieres, puedes copiar y pegar este fragmento de SQL en tu cliente de base de datos, o usa los pipes o tuberías de Unix (|) para pasarlo directamente. De todas formas, Django provee una manera más fácil de confirmar el envío del SQL a la base de datos.

Una vez creado las migraciones con makemigrations, es necesario sincronizar los cambios en la base de datos, ya que si ejecutas python manage.py makemigrations de nuevo nada sucede, porque no has agregado ningún modelo a la aplicación biblioteca ni has incorporado ninguna aplicación en INSTALLED_APPS. Por lo que, siempre es seguro ejecutar python manage.py makemigrations – no hará desaparecer, ni aparecer cosas mágicamente.

Ahora para realizar los cambios al esquema de la base de datos es necesario usar el comando mígrate, que se encarga de crear las tablas de la base de datos:

```
python manage.py migrate
```

Y muestra por salida algo así:

```
Operations to perform:
  Synchronize unmigrated apps: (none)
  Apply all migrations: biblioteca
Synchronizing apps without migrations:
  Creating tables...
  Installing custom SQL...
  Installing indexes...
Running migrations:
  Applying biblioteca.0001_initial... OK
```

El comando migrate es una simple sincronización de tus modelos hacia tu base de datos. Este comando examina todos los modelos en cada aplicación que figure en tu variable de configuración INSTALLED_APPS, verifica la base de datos para ver si las tablas apropiadas ya existen, y las crea si no existen.

El comando migrate toma todas las migraciones que se han aplicado al proyecto (ya que Django rastrea cada una de las migraciones aplicadas, usando una tabla especial llamada django_migrations), esencialmente las ejecuta de nuevo contra la base de datos, sincronizando los cambios hechos a los modelos con el esquema de la base de datos.

Las migraciones son muy poderosas y nos permiten cambiar los modelos cada cierto plazo de tiempo, como cuando estamos desarrollando nuestro proyecto, sin la necesidad de borrar las tablas o borrar la base de datos actual y crear otra – el propósito de las migraciones consiste en actualizar la base de datos que usamos, sin perder datos.

Los tres pasos que seguimos para crear cambios en el modelo.

1. Cambia tu modelo (en models.py).

2. Ejecuta python manage.py makemigrations para crear las migraciones para esos cambios.

3. Ejecuta python manage.py migrate para aplicar esos cambios a la base de datos.

La razón de usar comandos separados, para hacer y aplicar migraciones consiste en guardar las migraciones en un sistema de control de versiones y enviarlas con la aplicación, de esta forma el desarrollo será más fácil y también podrán ser usados por otros desarrolladores en producción.

Si estás interesado, toma un momento para bucear en el cliente de línea de comandos de tu servidor de bases de datos y ver las tablas que creó Django. Puedes ejecutar manualmente el cliente de línea de comandos (ej.: psql para PostgreSQL) o puedes ejecutar el comando python manage.py dbshell, que deducirá qué cliente de línea de comando ejecutar, dependiendo de tu configuración de servidor de base de datos. Esto último es casi siempre más conveniente.

Con todo esto, si estás interesado en verificar la base de datos, inicia el cliente de tu base de datos y ejecuta \dt (PostgreSQL), SHOW TABLES; (MySQL), o .schema (SQLite) para mostrar las tablas que Django ha creado.

Migraciones

En este punto, quizás te preguntes **¿Que son las migraciones?** Las migraciones son la forma en que Django se encarga de guardar los cambios que realizamos a los modelos (Agregando un campo, una tabla o borrando un modelo... etc.) en el esquema de la base de datos. Están diseñadas para funcionar en su mayor parte de forma automática, utilizan una versión de control para almacenar los cambios realizados a los modelos y son guardadas en un archivo del disco llamado "migration files", que no es otra cosa más que archivos Python, por lo que están disponibles en cualquier momento.

Las migraciones están creadas para funcionar sobre una aplicación Django, podemos pensar en ellas como en una versión de control para nuestra base de datos. Permiten a Django y a los desarrolladores manejar el esquema de la base de datos de forma transparente y duradera.

Existen dos comandos para usar e interactuar con las migraciones:

- **makemigrations**: es responsable de crear nuevas migraciones basadas en los cambios aplicados a nuestros modelos.
- **migrate**: responsable de aplicar las migraciones y los cambios al esquema de la base de datos.

Estos dos comandos se usan de forma interactiva, primero se crean o graban las migraciones, después se aplican:

```
python manage.py makemigrations

python manage.py migrate
```

Las migraciones se derivan enteramente de los archivos de los modelos y son esencialmente registros, que se guardan como historia, para que Django (o cualquier

desarrollador) pueda consultarlos, cuando necesita actualizar el esquema de la base de datos para que los modelos coincidan con los modelos actuales.

Acceso básico a datos

Una vez que has creado un modelo, Django provee automáticamente una API Python de alto nivel para trabajar con estos modelos. Prueba ejecutando el comando: python manage.py shell y escribiendo lo siguiente:

```
>>> from biblioteca.models import Editor
>>> p1 = Editor(nombre='Addison-Wesley', domicilio='75 Arlington Street',
...     ciudad='Boston', estado='MA', pais='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Editor(nombre="O'Reilly", domicilio='10 Fawcett St.',
...     ciudad='Cambridge', estado='MA', pais='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> Lista_Editores = Editor.objects.all()
>>> Lista_Editores
[<Editor: Editor object>, <Editor: Editor object>]
```

Estas pocas líneas logran bastantes resultados. Estos son los puntos sobresalientes:

- Para crear un objeto, sólo importa la clase del modelo apropiado y crea una instancia pasándole valores para cada campo.
- Para guardar el objeto en la base de datos, llama el método save() del objeto. Detrás de la escena, Django ejecuta aquí una sentencia SQL INSERT.
- Para recuperar objetos de la base de datos, usa el atributo Editor.objects. Busca una lista de todos los objetos Editor en la base de datos con la sentencia Editor.objects.all(). Detrás de escenas, Django ejecuta aquí una sentencia SQL SELECT.

Nota: Siempre guarda tus objetos con save():

Una cosa que vale la pena mencionar y que no fue muy clara en el ejemplo anterior, es que cuando creamos un objeto usando la API (la capa de modelo de Django), es que los objetos no se guardan en la base de datos, hasta que se llama al método save() explícitamente:

```
>>>p1 = Editor(...)# ¡En este punto, p1 no ha sido guardado en la base de datos!
>>>p1.save()# Ahora, si.
```

Si quieres crear y guardar un objeto en la base de datos, en un simple paso usa el método objects.create(). Este ejemplo, es equivalente al ejemplo anterior:

```
>>> p1 = Editor.objects.create(nombre='Apress',
...     domicilio='2855 Telegraph Avenue',
...     ciudad='Berkeley', estado='CA', pais='U.S.A.',
...     website='http://www.apress.com/')
```

```
>>> p2 = Editor.objects.create(nombre="O'Reilly",
...     domicilio='10 Fawcett St.', ciudad='Cambridge',
...     estado='MA', pais='U.S.A.',
...     website='http://www.oreilly.com/')
>>> Lista_Editores = Editor.objects.all()
>>> Lista_Editores
```

Naturalmente, puedes hacer mucho mas con la API de base de datos de Django –pero primero, arreglemos una pequeña incomodidad–.

Agrega cadenas de representación a tus modelos

Cuando imprimimos la lista de editores, todo lo que obtuvimos fue una salida poco útil, que hacía difícil distinguir los objetos Editor:

```
[<Editor: Editor object>, <Editor: Editor object>]
```

Podemos arreglar esto fácilmente agregando un método llamado __str__(), si estas usando python3 o un método __unicode__(), si estas usando Python2, a nuestro objeto Editor. Un método __str__() le dice a Python como mostrar la representación "string" de un objeto en unicode. Puedes ver esto en acción agregando un método __str__() a tus tres modelos:

biblioteca/models.py

```
from django.db import models

class Editor(models.Model):
    nombre = models.CharField(max_length=30)
    domicilio = models.CharField(max_length=50)
    ciudad = models.CharField(max_length=60)
    estado = models.CharField(max_length=30)
    pais = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self): # __unicode__ en Python 2
        return self.nombre

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField()

    def __str__(self): # __unicode__ en Python 2
        return '%s %s' % (self.nombre, self.apellidos)

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editor = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField()
    portada = models.ImageField(upload_to='portadas')

    def __str__(self): # __unicode__ en Python 2
        return self.titulo
```

Como puedes ver, un método __str__() puede hacer lo que sea que necesite hacer para devolver una representación textual. Aquí, los métodos __str__() de Editor y Libro devuelven simplemente el nombre y título del objeto respectivamente, pero el __str__() del Autor es un poco más complejo – junta los campos nombre y apellidos. El único requerimiento para __str__() es que devuelva una cadena. Si __str__() no devuelve una cadena si retorna, digamos, un entero – entonces Python generará un error, como TypeError con un mensaje como "__str__ returned non-string".

¿Que son los objetos Unicode? Puedes pensar en objetos Unicode, como en cadenas que pueden manejar más de un millón de distintos tipos de caracteres, que van desde versiones de caracteres latinos, no latinos, citas en chino, y símbolos "obscuros".

Las cadenas normales en Python2, son *codificadas* usando un tipo de codificación especial, tal como: ASCII, ISO-8859-1 o UTF-8 (En python3 todas las cadenas son Unicode). Si almacenas caracteres sencillos (cualquier cosa entre el estándar 128 ASCII, tal como letras de la A-Z y números del 0-9) en cadenas normales de Python2 no debes de perder de vista la codificación que estas usando, para que los caracteres puedan ser mostrados cuando sean imprimidos. Los problemas ocurren cuando guardamos los datos en un tipo de codificación y los combinamos con diferentes codificaciones, si tratamos de mostrarlos en nuestras aplicaciones, estas asumen un cierto tipo de codificación. Alguna vez has visto páginas Web o e-mails que muestran caracteres como "??? ??????" en lugar de palabras; esto generalmente sugiere un problema de codificación.

Los objetos Unicode no tienen una codificación, su uso es consistente, son un conjunto universal de caracteres llamado "Unicode." Cuando se utilizan objetos Unicode en Python, puedes mezclarlos y acoplarlos con seguridad, si tener que preocupare sobre problema de codificación.

Django utiliza objetos Unicode en todo el framework. Los objetos de los modelos son recuperados como objetos Unicode, las vistas interactúan con datos Unicode, y las plantillas son renderizadas como Unicode. Generalmente no debes preocuparte por esto, solo asegúrate que tus codificaciones sean correctas; y las cosas trabajaran bien.

Hemos tratado este tema muy a la ligera, sin embargo si quieres aprender más sobre objetos Unicode, un buen lugar para empezar es:

http://www.joelonsoftware.com/articles/Unicode.html

Para que los cambios sean efectivos, sal del shell Python (usando exit()) y entra de nuevo con python manage.py shell. (Esta es la manera más simple de hacer que los cambios en el código tengan efecto.) Ahora la lista de objetos Editor es más fácil de entender:

```
>>> from biblioteca.models import Editor
>>> ListaEditores = Editor.objects.all()
>>> ListaEditores
[<Editor: Addison-Wesley>, <Editor: O'Reilly>]
```

Asegúrate de que cada modelo que definas tenga un método __str__() – no solo por tu propia conveniencia cuando usas el intérprete interactivo, sino también porque Django usa la salida de __str__() en muchos otros lugares cuando necesita mostrar objetos.

¿Métodos __str__ o __unicode__?

En Python 3, es fácil, solo utiliza __str__().

En Python 2, es necesario definir métodos __unicode__() para que los valores que retornen sean unicode. Los modelos de Django usan por default __str__(), este método llama a __unicode__() y convierte el resultado a UTF-8 bytestring. Esto significa que unicode (p) retornara como cadena Unicode y str (p) devolverá una cadena normal, con los caracteres codificados como UTF-8. Python hace lo contrario: toma un objeto que tiene __unicode__ y llama a un método __str__ e interpreta el resultado como una cadena ASCII bytestring. Esta diferencia puede crear confusión.

Si todo esto no tiene sentido para ti, solo usa métodos str en Python 3 y todo funcionara bien.

Finalmente, observa como el metodo __str__() es un buen ejemplo para agregar *comportamiento* a los modelos. Un modelo Django describe más que la configuración de la tabla de la base de datos; también describe toda funcionalidad que el objeto sepa hacer. __str__() es un ejemplo de esa funcionalidad – un modelo sabe cómo mostrarse.

Insertar y actualizar datos

Ya has visto cómo se hace: para insertar una fila en tu base de datos, primero crea una instancia de tu modelo usando argumentos por nombre, así:

```
>>> p = Editor(nombre='Apress',
...     domicilio='2855 Telegraph Ave.',
...     ciudad='Berkeley',
...     estado='CA',
...     pais='U.S.A.',
...     website='http://www.apress.com/')
```

Este acto de instanciar una clase modelo *no* toca la base de datos.

Para guardar el registro en la base de datos (esto es, para realizar la sentencia SQL INSERT), llama al método save() del objeto:

```
>>> p.save()
```

En SQL, esto puede ser traducido directamente en lo siguiente:

```
INSERT INTO biblioteca_Editor
    (nombre, domicilio, ciudad, estado, pais, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA', 'U.S.A.', 'http://www.apress.com/');
```

Como el modelo Editor usa una clave primaria autoincremental id, la llamada inicial a save() hace una cosa más: calcula el valor de la clave primaria para el registro y lo establece como el valor del atributo id de la instancia:

```
>>> p.id
3 # esto es diferente, según tus datos
```

Las subsecuentes llamadas a save() guardarán el registro en su lugar, sin crear un nuevo registro (es decir, ejecutarán una sentencia SQL UPDATE en lugar de un INSERT):

```
>>> p.nombre = 'Apress Publishing'
>>> p.save()
```

La sentencia save() del párrafo anterior resulta aproximadamente en la sentencia SQL siguiente:

```
UPDATE biblioteca_Editor SET
    nombre = 'Apress Publishing',
    domicilio = '2855 Telegraph Ave.',
    ciudad = 'Berkeley',
    estado = 'CA',
    pais = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 3;
```

Ahora, observa como *todos* los campos han sido actualizados, no solamente el que habíamos cambiado. Dependiendo de tu aplicación, esto puede causar provocar una condición. Consulta como "Actualizar múltiples objetos en una sola declaración" más adelante, para ejecutar una consulta (ligeramente diferente).

```
UPDATE biblioteca_editor SET
    nombre = 'Apress Publishing'
WHERE id=3;
```

Seleccionar objetos

Conocer cómo crear y actualizar datos en una base de datos es esencial, pero hay ocasiones en que una aplicación web necesita realizar consultas para examinar los objetos que se han creado.

Ya hemos visto una forma de examinar *todos* los datos de un determinado modelo:

```
>>> Editor.objects.all()
[<Editor: Addison-Wesley>, <Editor: O'Reilly>, <Editor: Apress Publishing>]
```

Eso se traslada a esto; en SQL:

```
SELECT
    id, nombre, domicilio, ciudad, estado, pais, website
FROM book_Editor;
```

Nota que Django no usa SELECT * cuando busca datos y en cambio lista todos los campos explícitamente. Esto es una decisión de diseño: en determinadas circunstancias SELECT * puede ser lento, y (más importante) listar los campos sigue el principio del Zen de Python: "Explícito es mejor que implícito". Para saber más sobre el Zen de Python, intenta escribiendo import this en el prompt de Python.

Echemos un vistazo a cada parte de esta línea Editor.objects.all():

- En primer lugar, tenemos nuestro modelo definido, Editor. Aquí no hay nada extraño: cuando quieras buscar datos, usa el modelo para esto.

- Luego, tenemos objects. Técnicamente, esto es un administrador (manager). Los administradores son discutidos en el Capítulo10. Por ahora, todo lo que necesitas saber es que los administradores se encargan de realizar todas las operaciones a "nivel de tablas" sobre los datos incluidos, y lo más importante: **las consultas**.

- Todos los modelos automáticamente obtienen un administrador objects; debes usar el mismo cada vez que quieras consultar sobre una instancia del modelo.

- Finalmente, tenemos all(). Este es un método del administrador objects que retorna todas las filas de la base de datos. Aunque este objeto se parece a una lista, es realmente un QuerySet – un objeto que representa algún conjunto de filas de la base de datos. El Apéndice C describe QuerySets en detalle. Para el resto de este capítulo, sólo trataremos estos como listas emuladas.

Cualquier búsqueda en base de datos va a seguir esta pauta general – llamaremos métodos del administrador adjunto al modelo en el cual queremos hacer nuestra consulta.

Filtrar datos

Aunque obtener todos los objetos es algo que ciertamente tiene su utilidad, la mayoría de las veces lo que vamos a necesitar es manejarnos sólo un subconjunto de los datos. Para ello usaremos el método filter():

```
>>>Editor.objects.filter(nombre="Apress Publishing")
[<Editor: Apress Publishing>]
```

filter() toma argumentos clave que son traducidos en las cláusulas SQL WHERE apropiadas. El ejemplo anterior sería traducido en algo como:

```
SELECT
   id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_Editor
WHERE nombre = 'Apress Publishing';
```

Puedes pasarle a filter() múltiples argumentos, para reducir las cosas aún más:

```
>>>Editor.objects.filter(ciudad="Berkeley", estado="CA")
[<Editor: Apress Publishing>]
```

Estos múltiples argumentos son traducidos a cláusulas SQL AND. Por lo tanto el ejemplo en el fragmento de código se traduce a lo siguiente:

```
SELECT
   id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_Editor
WHERE ciudad = 'U.S.A.' AND estado = 'CA';
```

Observa que por omisión la búsqueda usa el operador SQL = para realizar búsquedas exactas. Existen también otros tipos de búsquedas:

```
>>> Editor.objects.filter(nombre__contains="press")
```

```
[<Editor: Apress Publishing>]
```

Nota el doble guión bajo entre nombre y contains. Del mismo modo que Python, Django usa el doble guión bajo para indicar que algo "mágico" está sucediendo – aquí la parte __contains es traducida por Django en una sentencia SQL LIKE:

```
SELECT
    id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_Editor
WHERE nombre LIKE '%press%';
```

Hay disponibles varios tipos de búsqueda, incluyendo icontains (LIKE no es sensible a diferencias de mayúsculas/minúsculas), startswith y endswith, y range (consultas SQL BETWEEN). El Apéndice C describe en detalle todos estos tipos de búsqueda.

Obtener objetos individuales

El ejemplo anterior usa el método filter() que retorna un QuerySet el cual es tratado como una lista, sin embargo en ocasiones desearás obtener un único objeto. Para esto existe el método get():

```
>>> Editor.objects.get(nombre="Apress Publishing")
<Editor: Apress Publishing>
```

En lugar de una lista (o más bien, un QuerySet), este método retorna un objeto individual. Debido a eso, una consulta cuyo resultado contenga múltiples objetos causará una excepción:

```
>>> Editor.objects.get(pais="U.S.A.")
Traceback (most recent call last):
...
MultipleObjectsReturned: get() returned more than one Editor --
    it returned 2! Lookup parameters were {'pais': 'U.S.A.'}
```

Por lo que una consulta, que no retorne un objeto también causará una excepción:

```
>>> Editor.objects.get(nombre="Pinguino")
Traceback (most recent call last):
...
DoesNotExist: Editor matching query does not exist.
    ...
```

La excepción DoesNotExist es un atributo del modelo de la clase Editor.DoesNotExist. Si quieres atrapar las excepciones en tus aplicaciones, puedes hacerlo de la siguiente forma:

```
try:
    p = Editor.objects.get(nombre='Apress')
except Editor.DoesNotExist:
    print ("Apress no está en la base de datos.")
else:
    print ("Apress está en la base de datos.")
```

Ordenar datos

A medida que juegas con los ejemplos anteriores, podrías descubrir que los objetos son devueltos en lo que parece ser un orden aleatorio. No estás imaginándote cosas, hasta ahora no le hemos indicado a la base de datos cómo ordenar los resultados, de manera que simplemente estamos recibiendo datos con algún orden arbitrario seleccionado por la base de datos.

Eso es, obviamente, un poco ingenuo. No quisiéramos que una página Web que muestra una lista de editores estuviera ordenada aleatoriamente. Así que, en la práctica, probablemente querremos usar order_by() para reordenar nuestros datos en listas más útiles:

```
>>> Editor.objects.order_by("nombre")
[<Editor: Addison-Wesley>, <Editor: Apress Publishing>, <Editor: O'Reilly>]
```

Esto no se ve muy diferente del ejemplo del método all() anterior, pero el SQL incluye ahora un ordenamiento específico:

```
SELECT
    id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_Editor
ORDER BY nombre;
```

Podemos ordenar por cualquier campo que deseemos:

```
>>> Editor.objects.order_by("domicilio")
[<Editor: O'Reilly>, <Editor: Apress Publishing>, <Editor: Addison-Wesley>]

>>> Editor.objects.order_by("estado")
[<Editor: Apress Publishing>, <Editor: Addison-Wesley>, <Editor: O'Reilly>]
```

Para ordenar por múltiples campos (donde el segundo campo es usado para quitar las ambigüedades en el orden, en casos donde el nombre sea el mismo), puedes usar múltiples argumentos:

```
>>> Editor.objects.order_by("estado", "domicilio")
[<Editor: Apress Publishing>, <Editor: O'Reilly>, <Editor: Addison-Wesley>]
```

También podemos especificar un ordenamiento inverso antecediendo al nombre del campo un prefijo - (el símbolo menos):

```
>>> Editor.objects.order_by("-nombre")
[<Editor: O'Reilly>, <Editor: Apress Publishing>, <Editor: Addison-Wesley>]
```

Aunque esta flexibilidad es útil, usar order_by() todo el tiempo puede ser demasiado repetitivo. La mayor parte del tiempo tendrás un campo en particular por el que usualmente desearás ordenar tus datos. Es esos casos Django te permite anexar al modelo un ordenamiento por omisión, usando una clase interna Meta:

```
class Editor(models.Model):
    nombre = models.CharField(max_length=30)
    domicilio = models.CharField(max_length=50)
    ciudad = models.CharField(max_length=60)
    estado = models.CharField(max_length=30)
    pais = models.CharField(max_length=50)
    website = models.URLField()

   class Meta:
       ordering = ["nombre"]

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

Hemos introducido un nuevo concepto: class Meta (Una clase Meta interna), esta clase esta embebida en la definición de la clase Editor``(identada dentro de la clase ``Editor). Podemos usar una clase Meta en cualquier modelo para especificar varias opciones específicas en un modelo. Examina el Apéndice B para conocer las opciones que puede poner bajo Meta. En el ejemplo anterior ordering = ["nombre"] le indica a Django que a menos que se proporcione un ordenamiento mediante order_by(), todos los editores deberán ser ordenados por su nombre.

Django usa esta class Meta interna como un lugar en el cual se pueden especificar metadatos adicionales acerca de un modelo. Es completamente opcional, pero puede realizar algunas cosas muy útiles, como usar el nombre en plural de la tabla usando verbose_name_plural.

Encadenar búsquedas

Has visto cómo puedes filtrar datos y has visto cómo ordenarlos. En ocasiones, por supuesto, vas a querer realizar ambas cosas. En esos casos simplemente "encadena" las búsquedas entre sí:

```
>>> Editor.objects.filter(pais="U.S.A.").order_by("-nombre")
[<Editor: O'Reilly>, <Editor: Apress Publishing>, <Editor: Addison-Wesley>]
```

Como podrías esperar, esto se traduce a una consulta SQL conteniendo tanto un WHERE como un ORDER BY:

```
SELECT id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_Editor
WHERE country = 'U.S.A'
ORDER BY nombre DESC;
```

Puedes encadenar consultas en forma consecutiva tantas veces como desees. No existe un límite para esto.

Rebanar datos

Otra necesidad común es buscar sólo un número fijo de filas. Imagina que tienes miles de editores en tu base de datos, pero quieres mostrar sólo el primero. Puedes hacer eso usando la sintaxis estándar de Python para el rebanado de listas:

```
>>> Editor.objects.all()[0]
<Editor: Addison-Wesley>
```

Esto se traduce, someramente, a:

```
SELECT
   id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_editor
ORDER BY nombre
LIMIT 1;
```

Similarmente, puedes recuperar un subconjunto específico de datos usando la sintaxis de Python y rebanando un rango de datos:

```
>>> Editor.objects.order_by('nombre')[0:2]
```

Esto devuelve dos objetos, lo que sería equivalente en SQL a:

```
SELECT id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_editor
ORDER BY nombre
OFFSET 0 LIMIT 2;
```

Observa que el rebanado negativo *no* está soportado:

```
>>> Editor.objects.order_by('nombre')[-1]
Traceback (most recent call last):
...
AssertionError: Negative indexing is not supported.
```

Sin embargo es fácil darle la vuelta a esto. Cambia la declaración order_by() así:

```
>>> Editor.objects.order_by('-nombre')[0]
```

Actualizar múltiples campos en una sola declaración

Ya vimos en la sección "*Insertando y actualizando datos*" que el método save() actualiza *todas* las columnas de una fila. Sin embargo, dependiendo de nuestra aplicación, podemos actualizar únicamente un subconjunto de columnas.

Por ejemplo, digamos que queremos actualizar el nombre Apress de la tabla Editor, para cambiarle el nombre de 'Apress' a 'Apress Publishing'. Usando el método save(), podemos hacerlo así:

```
>>> p = Editor.objects.get(nombre='Apress')
>>> p.nombre = 'Apress Publishing'
>>> p.save()
```

Esto se traduce, aproximadamente en la siguiente declaración SQL:

```
SELECT id, nombre, domicilio, ciudad, estado, pais, website
FROM biblioteca_libro
WHERE nombre = 'Apress';
```

```
UPDATE biblioteca_editor SET
    nombre = 'Apress Publishing',
    domicilio = '2855 Telegraph Ave.',
    ciudad = 'Berkeley',
   estado = 'CA',
   pais = 'U.S.A.',
   website = 'http://www.apress.com'
WHERE id = 3;
```

(Observa que el ejemplo asume que un Editor Apress, tiene un ID = 3)
Puedes ver en este ejemplo que el método save(), guarda *todos* los valores de las columnas, no solo la columna nombre. Si estas en un entorno donde otras columnas de la base de datos puedan cambiar debido a otro proceso, es más elegante cambiar *únicamente* una columna que se necesita cambiar. Para esto usa el método update() cuando consultes objetos. Por ejemplo:

```
>>> Editor.objects.filter(id=1).update(nombre='Apress Publishing')
```

La traducción a SQL es mucho más eficiente y no hay probabilidades de errores:

```
UPDATE biblioteca_editor
SET name = 'Apress Publishing'
WHERE id = 1;
```

El método update() trabaja con cualquier consulta, lo cual quiere decir que puedes editar múltiples registros a la vez. Esta es la forma en podemos cambiar ciudad de 'U.S.A.' a USA en cada registro Editor:

```
>>> Editor.objects.all().update(ciudad='USA')
2
```

El método update() retorna un valor – un entero que representa las veces que un registro ha cambiado. En el ejemplo anterior obtuvimos 2.

Borrar objetos

Para eliminar objetos, simplemente llama al método delete() de tu objeto:

```
>>> p = Editor.objects.get(nombre="Addison-Wesley")
>>> p.delete()
>>> Editor.objects.all()
[<Editor: Apress Publishing>, <Editor: O'Reilly>]
```

También puedes borrar objetos al por mayor llamando al método delete() en el resultado de cualquier consulta. Esto es similar a el método update() que mostramos en la sección anterior:

```
>>> Editor.objects.filter(ciudad='USA').delete()
>>> Editor.objects.all().delete()
>>> Editor.objects.all()
```

```
[]
```

Los borrados son *definitivos*, así que, **¡se cuidadoso!** Como medida de precaución, Django evita que puedas borrar los datos de una tabla en particular directamente, a menos que explícitamente así lo requieras, usa el método all(), si quieres borrar *toda* la tabla.

Por ejemplo esto no trabaja:

```
>>> Editor.objects.delete()
Traceback (most recent call last):
File "<console>", line 1, in <module>
AttributeError: 'Manager' object has no attribute 'delete'
```

Pero si agregas un método all(), funcionara bien:

```
>>> Editor.objects.all().delete()
```

Si solo quieres borrar un subconjunto de datos, no necesitas incluir el método all():

```
>>> Editor.objects.filter(ciudad='USA').delete()
```

¿Qué sigue?

Después de leer este capítulo, haz adquirido el conocimiento suficiente sobre los modelos Django, esto te permitirá escribir aplicaciones básicas usando una base de datos. En él *capítulo 10* encontraras más información sobre un uso más avanzado de la capa de modelos de Django.

Una vez que has definido tus modelos, el siguiente paso es ingresar datos a la base de datos. Podrías tener datos legados, en cuyo caso el *capítulo 18* te aconsejará acerca de cómo integrar bases de datos heredadas. Podrías delegar en el usuario del sitio la provisión de los datos, en cuyo caso el *capítulo 7* te enseñará cómo procesar datos enviados por los usuarios mediante formularios.

Pero en algunos casos, tú o tu equipo podrían necesitar ingresar datos en forma manual, en cuyo caso sería de ayuda el disponer de una interfaz basada en Web para el ingreso y el manejo de los datos. Él *próximo capítulo* está dedicado a la interfaz de administración de Django, la cual existe precisamente por esa razón.

CAPÍTULO 6

El sitio de administración

Para un cierto tipo de sitios Web, una **interfaz de administración** es una parte esencial de su infraestructura. Ya que se trata de una interfaz basada en HTML, limitada a los administradores autorizados, que permite agregar, editar y eliminar el contenido del sitio.

La interfaz que usas para escribir en tu blog, el sitio privado que los editores usan para moderar los comentarios de sus lectores, la herramienta que tus clientes utilizan para actualizar los comunicados de prensa en la Web, que construiste para ellos – todos ellos son ejemplos de interfaces de administración.

Sin embargo, existe un problema con las interfaces de administración: es aburrido construirlas. El desarrollo Web es divertido cuando estás desarrollando funcionalidades del lado público del sitio, pero construir interfaces de administración es siempre lo mismo. Tienes que autentificar usuarios, mostrar y manipular formularios, validar las entradas y demás tareas. Es aburrido y repetitivo.

¿Cuál es la solución de Django para estas tareas aburridas y repetitivas? Las hace todas por ti – en sólo un par de líneas de código, ni más ni menos. Con Django, construir interfaces de administración es un problema resuelto.

Este capítulo trata sobre la interfaz de administración automática de Django. Esta característica funciona leyendo las metadatos en tus modelos para brindar una interfaz potente y lista para producción que los administradores del sitio podrán usar inmediatamente. En este capítulo discutimos cómo activar, usar y personalizar esta utilidad.

Recomendamos leer este capítulo, inclusive si no te propones usar el sitio de administración, porque introduciremos algunos conceptos generales, que se pueden aplicar a Django en general, sin importar si usas la interfaz administrativa o no.

El paquete django.contrib

La interfaz de administración de Django es solo una parte de la gran suite de funcionalidades llamado django.contrib, la parte del código de Django que contiene diversos accesorios útiles del framework base. Puedes pensar en django.contrib como el equivalente a una librería estándar de Python – opcional, efectivo y que implementa muchos patrones comunes. Las cuales estas incrustados en Django y nos ayudaran a no reinventar la rueda en nuestras aplicaciones.

El sitio de administración es la primera parte de django.contrib, que cubriremos en este libro; técnicamente, es llamado django.contrib.admin. Algunas características más que podemos mencionar de django.contrib, es que incluye un sistema de autentificación (django.contrib.auth), ofrece soporte para sesiones anónimas (django.contrib.sessions), etiquetas para desarrolladores Web, (django.contrib.webdesign), etiquetas para darle un toque humano al framework (django.contrib.humanize).

En este capítulo conocerás diversas características de django.contrib que te convertirán en un experto en Django, discutiremos más adelante este tema, en el *capítulo16*. Por ahora, simplemente ten en cuenta que este paquete contiene muchos agregados útiles, y que django.contrib es generalmente el lugar en donde se localizan.

Activar la interfaz administrativa

Una de las partes más poderosas y atractivas de Django es la interfaz administrativa. Que se encarga de leer, automáticamente los metadatos de tus modelos para proveer una interfaz poderosa y lista para producción, para que inmediatamente puedas comenzar a añadirle contenido a tu sitio Web.

Sin embargo la interfaz de administración es enteramente opcional, porque únicamente un cierto tipo de sitios necesitan esta funcionalidad. Esto significa que se puede activar o desactivar según tus necesidades específicas.

En la siguiente parte discutimos cómo activar, usar y personalizar la interfaz administrativa de Django.

Para referencia estos son los requisitos:

- Agrega 'django.contrib.admin' a la variable INSTALLED_APPS de tu archivo de conImagenciones.(El orden en INSTALLED_APPS si importa, sin embargo estas están ordenadas alfabéticamente para facilitar su lectura, y Django las carga conforme están ordenadas, este detalle es importante ya que si quieres que django cargue primero tus plantillas, necesitas poner tu aplicación antes que el propio 'django.contrib.admin', para que las use.)

- La interfaz administrativa tiene cuatro dependencias (que dependen unas de otras), por lo que asegúrate que estén todas activadas:

  - django.contrib.auth
  - django.contrib.contenttypes
  - django.contrib.messages
  - django.contrib.sessions

  Si estas aplicaciones no estás listadas en la variable INSTALLED_APPS, agrégalas.

- Agrega django.contrib.messages.context_processors.messages a TEMPLATE_CONTEXT_PROCESSORS (a la variable) así como también agrega django.contrib.auth.middleware.AuthenticationMiddleware y django.contrib.messages.middleware.MessageMiddleware a MIDDLEWARE_CLASSES. (Estos están activados por omisión, así que solo necesitas hacerlo manualmente, si hiciste algún cambio a la configuración o los habías comentado previamente).

- Determina que modelos de tus aplicaciones serán editables en la interfaz administrativa.

  No todos los modelos pueden (o deberían) ser editables por los usuarios administradores, por lo que necesitas "marcar" los modelos que deberían tener una interfaz de administración. (Añadiendo una clase ModelAdmin al archivo admin.py.)

- Por cada uno de los modelos, crea opcionalmente una clase ModelAdmin en el archivo admin.py, que encapsule las funcionalidades personalizadas y las opciones específicas, para cada modelo en particular.

- Instancia una clase AdminSite y registra cada uno de los modelos en la clase ModelAdmin

- Apunta la instancia AdminSite a tu URLconf.

■**Nota:** La interfaz administrativa está habilitado por omisión en la plantilla de tu proyecto, si creaste tu proyecto usando startproject. La cual incluye una URL y una configuración para usar SQLite como base de datos. Por lo que solo debes preocuparte por los requisitos anteriores, si haz personalizado, borrado o comentado el archivo de configuración settings.py.

Una vez que nos hemos asegurado de tener todos los requisitos en orden, podemos llamar directamente al comando migrate para que se encargue de instalar las tablas, que la interfaz de administración necesita en la base de datos.

```
python manage.py migrate
```

Como se menciona en capítulos anteriores, el comando migrate examina todos los modelos en cada aplicación activada, que figure en tu variable de configuración INSTALLED_APPS, y verifica el esquema de la base de datos para comprobar si las tablas apropiadas ya existen, y las crea si no existen. Mostrando un mensaje por cada migración aplicada.

La interfaz administrativa instala 4 aplicaciones y una de ellas es django.contrib.auth el sitio de autorizaciones, por lo que al instalarlo es necesario crear interactivamente un súper-usuario, con el siguiente comando:

```
python manage.py createsuperuser
```

Sigamos interactivamente la salida del comando createsuperuser:

- Nos pedirá un nombre de usuario. Por defecto utilizara el nombre del sistema. Introducimos el nombre de nuestra preferencia y presionamos de nuevo enter:

  ```
  Username (leave blank to use 'your_username'): admin
  ```

- Nos pedirá también una dirección de correo electrónico::

  ```
  Email address: admin@example.com
  ```

- Finalmente nos pedirá una contraseña. Por lo que es necesario introducir dos veces la misma contraseña (la segunda vez solo como confirmación de la primera):

  ```
  Password: **********
  Password (again): *********
  Superuser created successfully.
  ```

El comando migrate toma todas las migraciones que se han aplicado al proyecto (ya que Django rastrea cada una de las migraciones aplicadas, usando una tabla especial llamada django_migrations), esencialmente las ejecuta de nuevo contra la base de datos, sincronizando los cambios hechos a los modelos con el esquema de la base de datos.

Una vez creadas las tablas, solo necesitas agregar el patrón URL al archivo urls.py. Si aún estás usando el que fue creado por startproject, el patrón de la URL de administración ya debería estar ahí, pero comentado. De cualquier forma, el patrón URL debe terminar viéndose así:

urls.py

```
from django.conf.urls import url
from django.contrib import admin

urlpatterns =[
    url(r'^admin/', include('django.contrib.admin.urls')),
]
```

Eso es todo. Ahora ejecuta python manage.py runserver para iniciar el servidor de pruebas. Verás algo como esto:

```
Validating models...
0 errors found.

Django version 1.8, using settings 'misitio.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Ahora puedes visitar la URL que te brinda Django, para acceder a la interfaz administrativa, identificarte, y jugar un poco. (http://127.0.0.1:8000/admin/)

Usar la interfaz de administración

La interfaz de administración está diseñada para ser usada por usuarios no técnicos, y como tal debería ser lo suficientemente clara como para explicarse por sí misma. Aún así, se brindan algunas notas sobre sus genialidades características.

Lo primero que verás será una página de identificación, como la que se muestra a continuación:

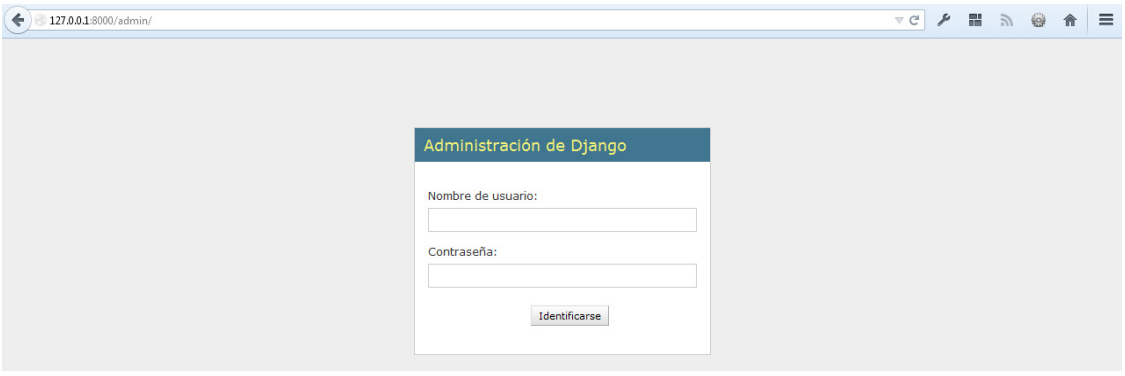


Imagen 6-1. Pantalla de autentificación de Django.

Usa el nombre de usuario y la clave que configuraste cuando agregaste el **superusuario** a través de la terminal. Una vez identificado, verás que puedes gestionar usuarios, grupos y permisos (veremos más sobre esto en breve).

Una vez que nos hemos autentificado, la primera cosa que veremos será la página de inicio o "índice". Esta página contiene una lista de todos los datos disponibles que pueden ser editados en la página del sitio de administración. Como aun no hemos activado nuestros modelos (Los activaremos más adelante), la lista de aplicaciones es escasa, solo incluye **Grupos** y **Usuarios**, los cuales son agregados a la interfaz por omisión.

Imagen 6-2. El índice principal de la Administración de Django.

Cada tipo de datos en la interfaz administrativa, contienen enlaces para agregar y modificar objetos, que nos llevan a páginas especificas a las que nos referiremos como listas de cambio y formularios de edición de objetos: La lista de cambios muestra todos los objetos disponibles en la base de datos, mientras que el formulario de edición nos permite agregar, cambiar o borrar registros específicos de la base de datos.

Otros lenguajes en Django: Si tu lenguaje natural, no es el ingles y si tu navegador está configurado para aceptar otros lenguajes además del ingles, puedes hacer un cambio rápidamente para ver la interfaz administrativa traducida a tu idioma. Solo agrega a la variable LANGUAGE_CODE, que se encuentra en el archivo de configuraciones tu idioma nativo. Django cuenta con traducciones para muchos lenguajes, entre los que se encuentran al momento de escribir esto:

'af', 'Afrikaans', 'ar', 'Arabic', 'az', 'Azerbaijani', 'bg', 'Bulgarian', 'be', 'Belarusian', 'bn', 'Bengali', 'br', 'Breton', 'bs', 'Bosnian', 'ca', 'Catalan', 'cs', 'Czech', 'cy', 'Welsh', 'da', 'Danish', 'de', 'German', 'el', 'Greek', 'en', 'English', 'en-au', 'Australian English', 'en-gb', 'British English', 'eo', 'Esperanto', 'es', 'Spanish', 'es-ar', 'Argentinian Spanish', **'es-mx', 'Mexican Spanish',** 'es-ni', 'Nicaraguan Spanish', 'es-ve', 'Venezuelan Spanish', 'et', 'Estonian', 'eu', 'Basque', 'fa', 'Persian', 'fi', 'Finnish', 'fr', 'French', 'fy', 'Frisian', 'ga', 'Irish', 'gl', 'Galician', 'he', 'Hebrew', 'hi', 'Hindi', 'hr', 'Croatian', 'hu', 'Hungarian', 'ia', 'Interlingua', 'id', 'Indonesian', 'is', 'Icelandic', 'it', 'Italian', 'ja', 'Japanese', 'ka', 'Georgian', 'kk', 'Kazakh', 'km', 'Khmer', 'kn', 'Kannada', 'ko', 'Korean', 'lb', 'Luxembourgish', 'lt', 'Lithuanian', 'lv', 'Latvian', 'mk', 'Macedonian', 'ml', 'Malayalam', 'mn', 'Mongolian', 'my', 'Burmese', 'nb', 'Norwegian Bokmal', 'ne', 'Nepali', 'nl', 'Dutch', 'nn', 'Norwegian Nynorsk', 'os', 'Ossetic', 'pa', 'Punjabi', 'pl', 'Polish', 'pt', 'Portuguese', 'pt-br', 'Brazilian Portuguese', 'ro', 'Romanian', 'ru', 'Russian', 'sk', 'Slovak', 'sl', 'Slovenian', 'sq', 'Albanian', 'sr', 'Serbian', 'sr-latn', 'Serbian Latin', 'sv', 'Swedish', 'sw', 'Swahili', 'ta', 'Tamil', 'te', 'Telugu', 'th', 'Thai', 'tr', 'Turkish', 'tt', 'Tatar', 'udm', 'Udmurt', 'uk', 'Ukrainian', 'ur', 'Urdu', 'vi', 'Vietnamese', 'zh-cn', 'Simplified Chinese', 'zh-hans', 'Simplified Chinese', 'zh-hant', 'Traditional Chinese', 'zh-tw', 'Traditional Chinese'

Solo agrega el código de tu lenguaje así:

```
LANGUAGE_CODE = 'es-mx'
```

También agrega 'django.middleware.locale.LocaleMiddleware' a la variable MIDDLEWARE_CLASSES del archivo de configuraciones, solo asegúrate de que aparezca *después* de 'django.contrib.sessions.middleware.SessionMiddleware'.

Una vez hecho esto, recarga la página de índice de la interfaz administrativa. Si está disponible alguna traducción para tu lenguaje, entonces varias partes de la interfaz –como "cambiar contraseña", "cerrar sesión", enlaces que se encuentran en la parte superior de la pagina, aparecerán en tu idioma.

Para conocer más características basadas en internacionalización, puedes consultar el *capítulo19*,

Da clic en el link "Usuarios" en la fila de "Usuarios" para ingresar a la página de lista de usuarios registrados.

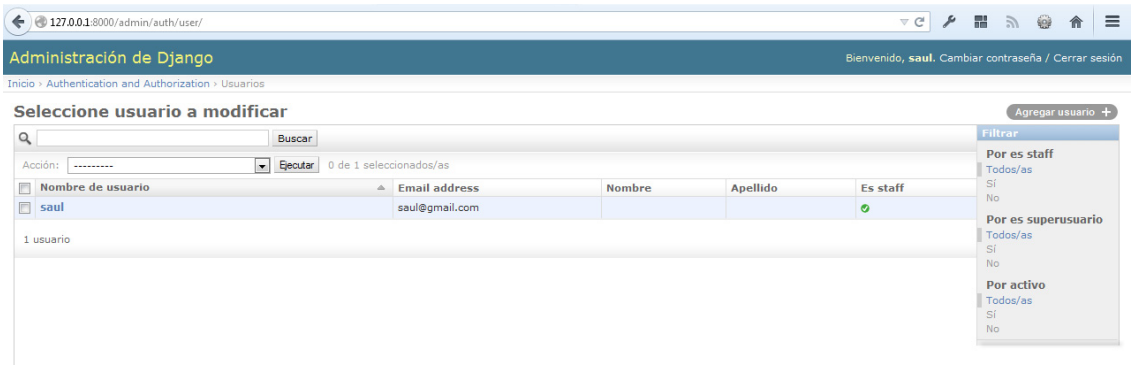


Imagen 6-3. La lista de cambios de usuarios.

Esta página muestra todos los usuarios de la base de datos, puedes pensar en ella como en una versión estilizada de una consulta SQL: SELECT * FROM auth_user; Si estas siguiendo estos ejemplos, asumiremos que solo haz agregado un usuario, sin embargo una vez que agregues mas usuarios, es probable que encuentres útiles las opciones para filtrar, ordenar o buscar. Las opciones para filtrar están en el lado derecho, las opciones para ordenar están disponibles dando clic en la cabecera de la columna y la caja de búsqueda está situada en la parte superior y te permitirán buscar usuarios por su nombre.

Da clic en el nombre de un usuario que hayas creado, para editarlo.

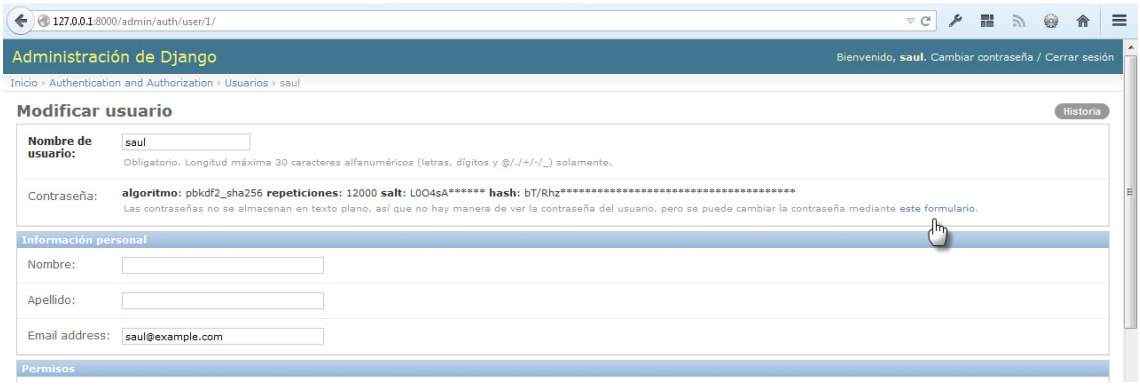


Imagen 6-4. Un formulario para editar usuarios.

Esta página te permite cambiar los atributos de un usuario, tal como el nombre, los apellidos y los distintos permisos. (Observa que para cambiar la contraseña de un usuario, es necesario dar clic en el formulario "cambiar contraseña", en el link "este formulario", debajo del campo contraseña, para cambiar el código hash.) Otra cosa que debes notar es que los distintos campos, utilizan diferentes widgets –Por ejemplo el campo fecha/tiempo tiene controles como un calendario y un reloj, los campos booleanos tienen checkboxes, los campos de caracteres tienen una simple caja de entrada de texto.

Puedes eliminar un registro, dando clic en el botón borrar, que se encuentra en el lado izquierdo del formulario. La interfaz de administración solicita una confirmación para prevenir errores. La eliminación de un objeto se desencadena en cascada, y la página de confirmación de eliminación del objeto muestra todos los objetos relacionados que se eliminarán con él. (Por ejemplo, si borras un Editor; cualquier libro que pertenezca a ese editor será borrado también.)

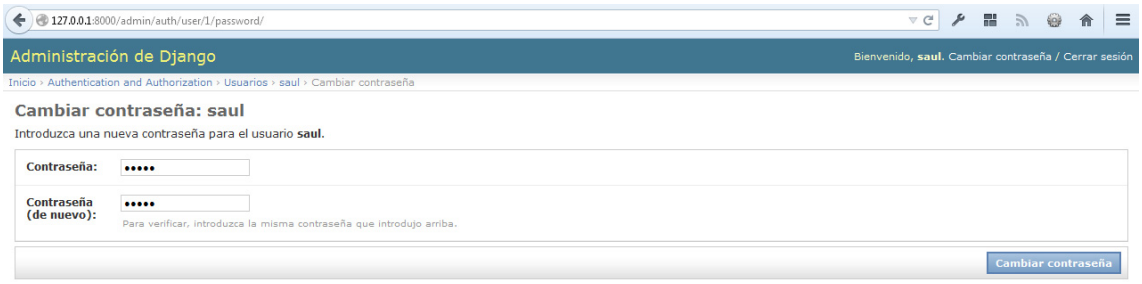


Imagen 6-5. Un formulario para cambiar contraseña de usuario.

Puedes agregar un nuevo usuario, dando clic en "Agregar" en la columna correspondiente, en la página de inicio de la interfaz administrativa. Esto te llevara a una página vacía, lista para que la rellenes.

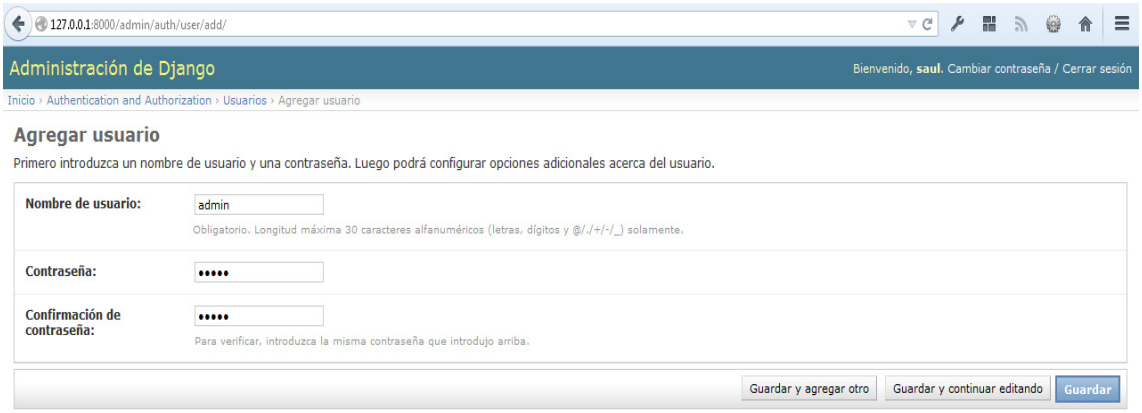


Imagen 6-6. Un formulario de edición para agregar un usuario.

Te darás cuenta que la interfaz de administración también controla por ti la validez de los datos ingresados. Intenta dejar un campo requerido (los cuales aparecen con letras en negritas) en blanco o poner una fecha inválida en un campo tipo fecha y verás los avisos resaltados en rojo, cuando intentes guardar el objeto, como se muestra en la Imagen siguiente:

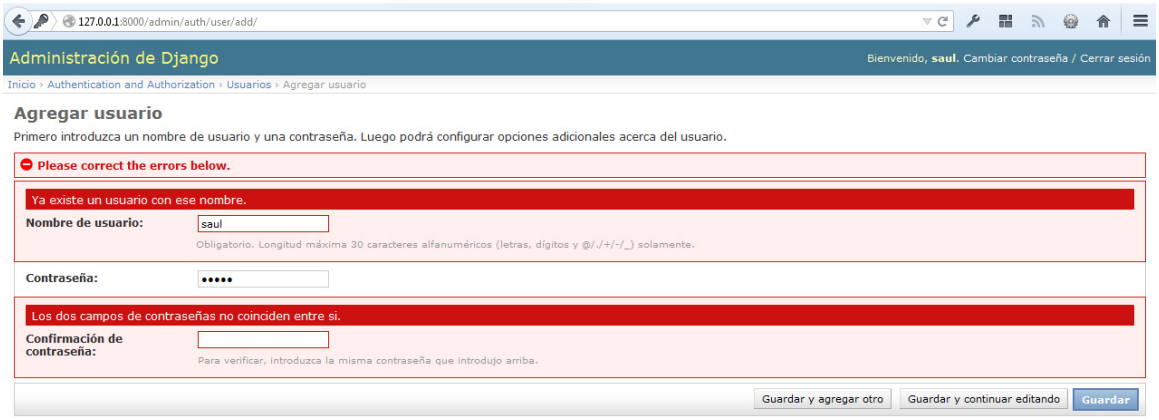


Imagen 6-7. Un formulario de edición mostrando errores de validación.

Cuando editas un objeto existente, verás el botón "Historia" en la esquina superior derecha de la ventana. Cada cambio realizado a través de la interfaz de administración es registrado, y puedes examinar este registro haciendo clic en este botón (mira la Imagen 6-8).

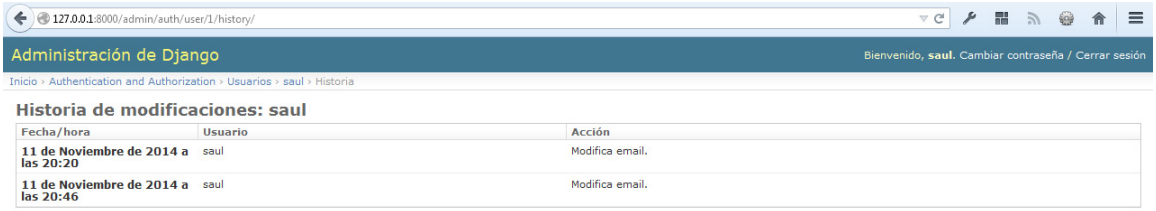


Imagen 6-8. Página de historia de un objeto en Django.

Agrega tus modelos al sitio administrativo

Hay una parte crucial que no hemos hecho todavía. Y es agregar nuestros modelos a la interfaz administrativa, para poder agregar, cambiar y borrar objetos en las tablas de la base de datos usando una interfaz agradable. Continuando con el ejemplo del *capítulo 5,* previamente definimos en nuestra aplicación biblioteca, tres modelos: Editor, Libro y Autor.

Dentro del directorio interno biblioteca (misitio/biblioteca), existe un archivo vacio llamado admin.py, creado automáticamente por el comando startapp agreguémosle las siguientes líneas de código, para registrar nuestros tres modelos:

biblioteca/admin.py

```
from django.contrib import admin
from biblioteca.models import Editor, Autor, Libro

admin.site.register(Editor)
admin.site.register(Autor)
admin.site.register(Libro)
```

Este código registra, cada uno de los modelos en la interfaz administrativa, para que Django nos ofrezca una interfaz para cada uno de los modelos registrados, bajo el nombre de la aplicación y podamos introducir datos directamente en ellos.

Una vez que hemos hecho esto, podemos navegar a la página de inicio usando un navegador Web en: http://127.0.0.1:8000/admin/, y podremos ver una sección

llamada “Biblioteca” con enlaces para Autor, Libros y Editores. (Si estabas ejecutando el servidor de desarrollo, es necesario detenerlo e iniciarlo de nuevo para que los cambios surtan efecto.)

Django usa el nombre de cada uno de los modelos, para presentarlos en la interfaz administrativa, sin embargo agrega la letra “s” para mostrar el nombre en plural, lo cual no siempre es lo más adecuado, si quieres mostrar el nombre en plural usa la opción verbose_name_plural en cada clase Meta interna, a la que le quieras agregar un nombre en plural. Agrégale uno a cada modelo así:

```
class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField()

    class Meta:
        ordering = ["nombre"]
        verbose_name_plural = "Autores"

    def __str__(self):              # __unicode__ en Python 2
        return '%s %s' % (self.nombre, apellidos)
```

Ahora tienes una completa interfaz administrativa funcional para cada uno de tus tres modelos. **¡Eso es genial no crees!**

Imagen 6-9. Página de inicio de la interfaz administrativa.

Tomate un momento para agregar, cambiar e insertar algunos registros mas en tu base de datos. Si estas siguiendo los ejemplos, en especial los del *capítulo5* cuando agregamos objetos mediante la terminal al modelo Editor (y no los borraste), puedes ver esos registros en la página de listado de editores.

Una característica que vale la pena mencionar, es que el sitio administrativo maneja las relaciones foráneas y las relaciones muchos a muchos, las cuales aparecen en el modelo Libro, si recuerdas así es como definimos el modelo:

```
class Libro(models. Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editores = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField()

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

Observa que el sitio de administración, contiene una página “Agregar libro” ubicada en: http://127.0.0.1:8000/admin/biblioteca/libro/add/, donde tenemos distintos tipos de campos como el de los editores (una “relación foránea”

ForeignKey), la cual es representada por una caja de selección y tenemos el campo autores (Una relación "Muchos a muchos" ManyToManyField), la cual es representada por una caja de selección múltiples. Ambos campos están situados al lado de un icono verde que permite agregar registros a las relaciones. Por ejemplo si das clic en el icono verde (un signo de "mas") en un campo de "editores" veras una ventana flotante que te permitirá agregar un editor. Después de que hayas creado satisfactoriamente un editor en la ventana flotante, el formulario "Agregar libro" mostrara una actualización con el nuevo editor creado.

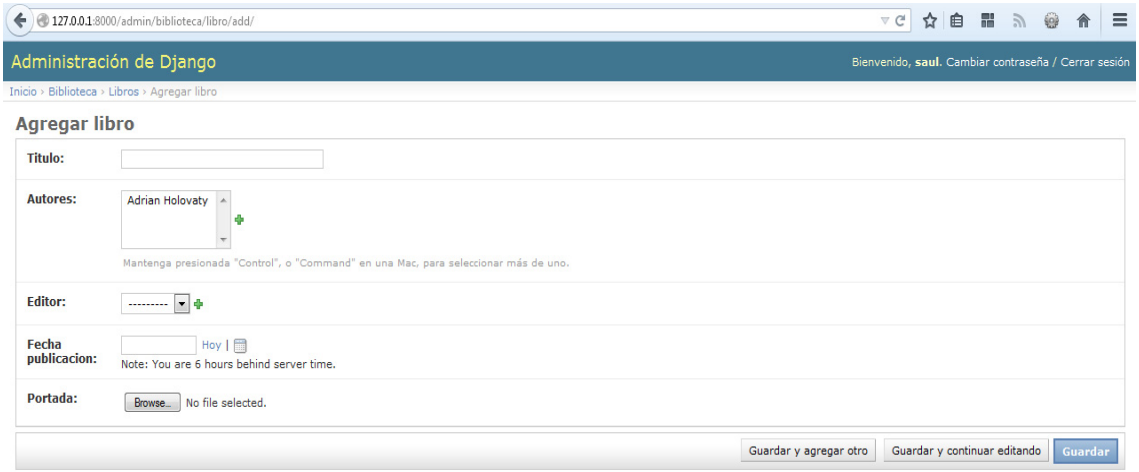


Imagen 6-10. Formulario para agregar libros, en la interfaz administrativa.

Como trabaja la interfaz administrativa

Detrás de escena, la forma en que trabaja la interfaz administrativa, es bastante directa.

Al iniciar el servidor, Django carga tus URLconf de urls.py y ejecuta la declaración admin.autodiscover() (que es habilitada por omisión), la cual se encarga de activar la interfaz administrativa. Esta función itera sobre cada una de las aplicaciones listadas en INSTALLED_APPS y busca un archivo llamado admin.py en cada una de las aplicaciones instaladas. Si existe un archivo admin.py, ejecuta el código del archivo. (Django automáticamente busca un modulo admin en cada una de las aplicaciones y lo importa.)

En el archivo admin.py de nuestra aplicación biblioteca, cada una de las llamadas a admin.site.register() simplemente registra cada uno de los modelos en la interfaz administrativa.

De esta forma el sitio administrativo mostrara una interfaz, que nos permitirá editar/cambiar cada uno de los modelos que hayamos explícitamente registrado.

La aplicación django.contrib.auth incluye su propio archivo admin.py, por lo que tanto Usuarios y Grupos aparecieron automáticamente en la interfaz administrativa. Otras aplicaciones de django.contrib, tal como django.contrib.redirects también pueden agregarse, así como muchas de las aplicaciones de terceros que descarguemos de la Web.

Más allá de esto, la interfaz administrativa es solo una aplicación Django, la cual incluye sus propios modelos, plantillas, vistas y patrones URL. Puedes agregarla a tus aplicaciones simplemente anclándola a tus URLconfs, tal como lo harías con una vista. Puedes inspeccionar sus plantillas, vistas y patrones URL, las cuales se encuentran en django/contrib/admin, en la copia de tu código base – Pero no intentes cambiar nada directamente, ya que existen otras formas para cambiar y personalizar la manera en que trabaja el sitio administrativo. (Si decides hurgar en la interfaz administrativa ten en cuenta que esta realiza, una gran cantidad de cosas

bastante complicadas leyendo los metadatos de los modelos, así que probablemente te tomara un buen tiempo leer y comprender el código.)

Como crear campos opcionales

Después de jugar un rato con el sitio de administración, probablemente encuentres algunas limitaciones – las formas para editar requieren que todos los campos sean completados, sin embargo en algunos casos es necesario que algunos campos sean opcionales. Digamos por ejemplo, que queremos que un modelo Autor contenga un campo email que sea opcional – es decir que permita cadena en blanco. En el mundo real, un autor puede no tener una dirección de correo electrónico o email .

Para especificar que un campo email sea opcional, edita el modelo Autor (el cual creamos en el *capítulo5*, y que se encuentra en misitio/biblioteca/models.py). Simplemente agrega **blank=True** al campo email así:

```
class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField(blank=True)
```

Esto le dice a Django que los valores en blanco están permitidos en el campo email de la tabla Autor. Por omisión, todos los campos se asignan con **blank=False**, lo cual no permite valores en blanco.

Veamos algunas cosas interesantes que pasan aquí: hasta ahora con la excepción del método __str__(), nuestros modelos ha servido como definiciones para las tablas de la base de datos –Esencialmente expresiones Pythonicas de la sentencia CREATE TABLE de SQL. Agregando blank=True, hemos comenzado a desplegar nuestro modelo más allá de una simple definición de tablas de nuestra base de datos. Ahora nuestro modelo de clases empieza a ser una rica colección de definiciones acerca del objeto Autor y lo que puede hacer. No únicamente es el campo email que representa una columna VARCHAR en la base de datos, es también un campo opcional dentro del contexto de la interfaz administrativa.

Una vez que hemos agregado blank=True, recarga el formulario "Agregar autor" en: http://127.0.0.1:8000/admin/biblioteca/autor/add/ y podrás darte cuenta que la etiqueta –"Email"– ya no está en negritas. Esto significa que el campo ya no es requerido. Podemos agregar ahora autores sin necesidad de proveer una dirección de email; por lo que ya no veremos el campo marcado de rojo que nos dice el mensaje "Este campo es requerido", ya que ahora podemos dejar el campo vacio.

Como crear campos numéricos y de fechas opcionales

Un problema común relacionado con campos en blancos o "blank=True" tiene que ver con los campos numéricos y de fechas, este tema requiere un poco de explicación a fondo.

SQL tiene sus propias maneras de especificar los valores en blanco – un valor especial llamado NULL. NULL significa "desconocido" o "no valido" u algún otro significado especifico.

En SQL un valor NULL es diferente que una cadena vacía, tal como el objeto especial Python None, que es diferente a una cadena vacía en Python (""). Esto significa que es posible que un campo de caracteres particular (por ejemplo una columna VARCHAR) contenga ambos valores: NULL y una cadena vacía.

Esto puede causar cierta ambigüedad y confusión, ¿"Porque este registro tiene un valor NULL y este otro una cadena vacía"? ¿Existe una diferencia o fueron los datos

registrados inconsistentemente? y como obtengo todos los registros que tienen un valor en blanco – busco ambos registros o únicamente selecciono las cadenas vacía.

Para ayudar a evitar estas ambigüedades, Django automáticamente genera un declaración CREATE TABLE (Que cubrimos en el capítulo 5) y agrega explícitamente en cada columna una definición NOT NULL. Por ejemplo esta es la declaración generada por el modelo Autor del capítulo 5:

```
CREATE TABLE "biblioteca_autor" (
    "id" serial NOT NULL PRIMARY KEY,
    "nombre" varchar(30) NOT NULL,
    "apellidos" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
);
```

En la mayoría de los casos, este comportamiento por omisión es optimo para usarlo en nuestras aplicaciones y nos ayudara a guardar nuestros datos y evitar inconsistencias y dolores de cabeza, tal como el sitio de administración que inserta cadenas vacías (no valores NULL) cuando dejamos un campo de caracteres en blanco.

Pero existe una excepción con algunas columnas de la base de datos que no aceptan cadenas vacías – por ejemplo los de tipo fechas, y los numéricos. Si intentas insertar una cadena vacía en una columna de tipo fecha o número entero, solo conseguirás un error de la base de datos, dependiendo de la base de datos que estés utilizando (PostgreSQL es estricta y lanzara una excepción; MySQL puede aceptar o no dependiendo de la versión que estés usando, el tiempo, el día y la fase de la luna) En este caso NULL es únicamente una forma de especificar que el valor esta vacio. En los modelos de Django, puedes especificar NULL agregando null=True a los campos, donde sea necesario.

De modo que existe una manera más larga de decir esto: si quieres permitir valores en blanco en un campo (por ejemplo: DateField, TimeField, DateTimeField) o numerico (por ejemplo: IntegerField, DecimalField, FloatField), necesitas agregar ambos tipos: **null=True** y **blank=True.**

Para ejemplificar mejor lo anterior, cambiemos el modelo Libro para que el campo fecha_publicacion permita espacios en blanco y valores nulos. Este es el código revisado:

```
class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editores = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField(blank=True, null=True)
    portada = models.ImageField(upload_to='portadas')
```

Agregar null=True es más complicado que agregar blank=True, porque null=True cambia la semántica de la base de datos – esto es, cambia la declaración CREATE TABLE para que remueva del campo fecha_publicacion, la declaración NOT NULL de la base de datos –. Para completar este cambio necesitamos actualizar el esquema de la base de datos.

En versiones anteriores de Django para actualizar la base de datos, necesitábamos manualmente usar el interprete de comandos (especifico de cada base de datos) y utilizar SQL para alterar el esquema de la base de datos, una vez que habíamos sincronizado nuestros modelos. Sin embargo esto ya no es necesario (aunque hay sus excepciones), ya que podemos usar las migraciones para realizar estas tareas, recuerdas los tres pasos que seguimos para instalar los modelos:

- Cambia tu modelo (en models.py).
- Ejecuta manage.py makemigrations para crear las migraciones para esos cambios.
- Ejecuta manage.py migrate para aplicar esos cambios a la base de datos.

Cada vez que cambiemos nuestros modelos, es necesario ejecutar estos dos comandos para sincronizar los cambios en el esquema de la base de datos automáticamente.

Una vez que hayamos creado las migraciones, sincronizando los modelos (para agregar valores nulos y campos en blanco), traemos de vuelta la interfaz administrativa, ahora el formulario "Agregar libro" permite publicar valores vacios en el campo fecha_publicacion y lo mejor de todo es que no tenemos que ejecutar SQL directamente.

Personalizar las etiquetas de los campos

En los formularios del sitio de administración, cada etiqueta de texto es generada, de cada uno de los nombres de cada campo. El algoritmo es simple: Django reemplaza los guiones bajos con espacios y pone en mayúscula la primera letra de la palabra. Así que por ejemplo, en el modelo libro, el campo fecha_publicacion tiene la etiqueta "Fecha publicación"

De cualquier manera, el nombre de los campos no siempre se presenta de una forma agradable en las etiquetas de texto, en algunos casos lo más recomendable es personalizar la etiqueta. Para hacerlo es necesario especificarlo con la etiqueta verbose_name en el campo del modelo.

Por ejemplo, así es como podemos cambiar la etiqueta del campo Autor email a "e-mail" con un guion en medio:

```
class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')
```

Para que los cambios surtan efecto, recarga el servidor y podrás ver la nueva etiqueta de texto e-mail, en el formulario para editar autores.

Observa que no necesitas poner en mayúscula la primera letra de la palabra cuando utilizas verbose_name ya que esta *siempre* será mostrada con la primera palabra en mayúsculas, a menos de que ha si lo requieras (por ejemplo: "USA estate"). Django automáticamente usara la mayúscula cuando lo necesite y mostrara la salida exacta del valor verbose_name en otros lugares que no requieran mayúsculas.

Finalmente, nota que puedes pasar **argumentos posicionales** a verbose_name, para lograr una sintaxis ligeramente más compacta. Este ejemplo es equivalente al anterior:

```
class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField('e-mail', blank=True)
```

Aunque ten en cuenta, que esto no trabaja con campos ManyToManyField o ForeignKey, porque estas relaciones requieren como primer argumento un modelo de clase. En estos casos, es necesario especificar explícitamente verbose_name de la forma habitual.

Clases personalizadas de la interfaz administrativa

Los cambios que hemos realizado hasta ahora blank=True, null=True y verbose_name son realmente a nivel de modelos, no a nivel administrativo. Es decir estos cambios son fundamentalmente una parte del modelo y solo ocurren cuando usamos el sitio administrativo, por lo que no hay nada especifico acerca de ellos.

Más allá de esto, el sitio administrativo ofrece abundantes opciones que te permiten modificar la forma en que el sitio administrativo trabaja para determinados modelos. Estas opciones se encuentran en las clases **ModelAdmin,** que son las clases que contienen la configuración específica para un modelo, de una instancia del sitio administrativo.

Personalizar la lista de cambios

Vamos a sumergirnos en la personalización de la interfaz administrativa, especificando que campos serán mostrados en la lista de cambios del modelo Autor.

Por omisión, la lista de cambios sólo muestra la cadena de representación del modelo que agregamos con el método __str__, en el capítulo5, definimos un método __str__ para el objeto Autor que muestra el primer nombre y los apellidos juntos así:

```
class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')

    def __str__(self):
        return '%s %s' % (self.nombre, self.apellidos)
```

Como consecuencia de estos cambios, la lista de cambios para el objeto Autor muestra juntos el nombre y los apellidos, como puedes ver en la Imagen 6-11.

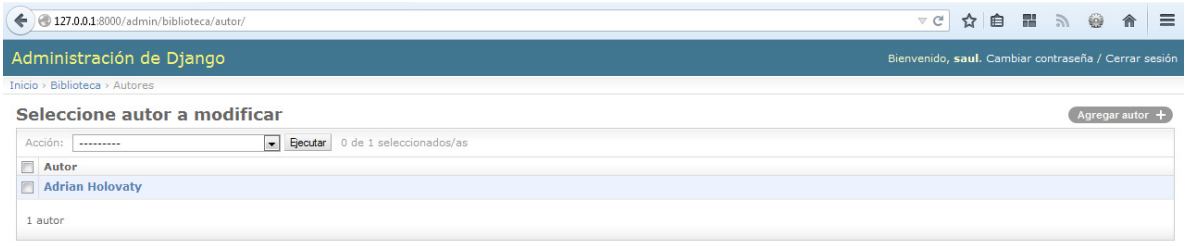


Imagen 6-11. Página de lista de cambios del modelo Autor.

Podemos pulir este comportamiento predeterminado agregando algunos campos más a la lista de cambios. Sería conveniente, por ejemplo ver el correo electrónico de cada autor y seria agradable poder ordenarlos por nombres y apellidos.

Para hacer que esto suceda, necesitamos definir una clase ModelAdmin para el modelo Autor. Esta clase es la clave para personalizar la interfaz administrativa y una de las cosas básicas que nos permite hacer, es especificar la lista de campos que queremos visualizar en la lista de cambios.

Edita el archivo admin.py para realizar estos cambios.

```
from django.contrib import admin
from biblioteca.models import Editor, Autor, Libro
```

```
class AutorAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'apellidos', 'email')

admin.site.register(Editor)
admin.site.register(Autor, AutorAdmin)
admin.site.register(Libro)
```

Esto es lo que hicimos:

- Creamos la clase AutorAdmin. Esta clase, la cual es una subclase de django.contrib.admin.ModelAdmin, se encarga de llevar a cabo la configuración para un modelo especifico de la interfaz administrativa. Únicamente especificamos una personalización –list_display, la cual es una tupla de nombres de campos, que controla que columnas aparecen en la lista de cambios. Siempre y cuando estos nombres de campos, existan en el modelo.

- Alteramos la llamada a admin.site.register(), para agregar AuthorAdmin después de Autor.

  Puedes leer esto como: "Registra el modelo Autor con las opciones de AuthorAdmin."
  La función admin.site.register() toma un subclase ModelAdmin como un segundo argumento opcional. Si no necesitas especificar un segundo argumento (como en el caso de Editor y de Libro) Django usara por omisión las opciones administrativas para el modelo.

Con estos cambios realizados, recarga la lista de cambios de autor y ahora podrás ver tres columnas –Nombre, Apellidos y E-mail. En suma, cada uno de estas columnas se puede ordenar dando clic en la cabecera de la columna. (Ve la Imagen 6-12)

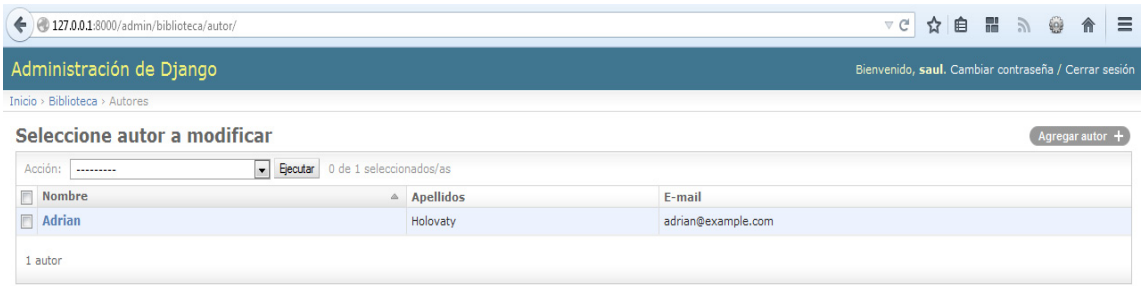


Imagen 6-12. La pagina de lista de cambios de autor, después de usar "list_display"

Ahora, agreguémosle una barra de búsqueda. Agrega search_fields a la clase AutorAdmin, así:

```
class AutorAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'apellidos' 'email')
    search_fields = ('nombre', 'apellidos')
```

Recarga la pagina en tu navegador y podrás observar una barra de búsqueda en la parte superior (Observa la Imagen 6-9.). Acabamos de informarle a la lista de cambios que incluya una barra de búsqueda, que se encargue de buscar en los campos nombre y apellidos de la base de datos. Como cualquier usuario pudiera esperar, no

distingue entre mayúsculas y minúsculas y busca en ambos campos, así que si buscamos la palabra "bar" podríamos encontrar un autor llamado “Barney” y también otro autor cuyo apellido sea “Hobarson”.

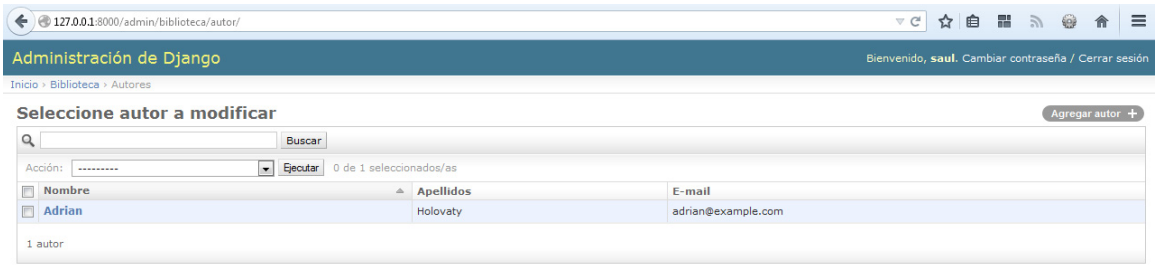


Imagen 6-13. Lista de cambios, después de agregar “search_fields”.

Después, podemos agregar algunos filtros al modelo Libro para mostrar la lista de cambios, por fechas de publicaciones:

```
from django.contrib import admin
from biblioteca.models import Editor, Autor, Libro

class AutorAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'apellidos', 'email')
    search_fields = ('nombre', 'apellidos')

class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)

admin.site.register(Editor)
admin.site.register(Autor, AutorAdmin)
admin.site.register(Libro, LibroAdmin)
```

Ahora visita la pagina de lista de cambios de libros:

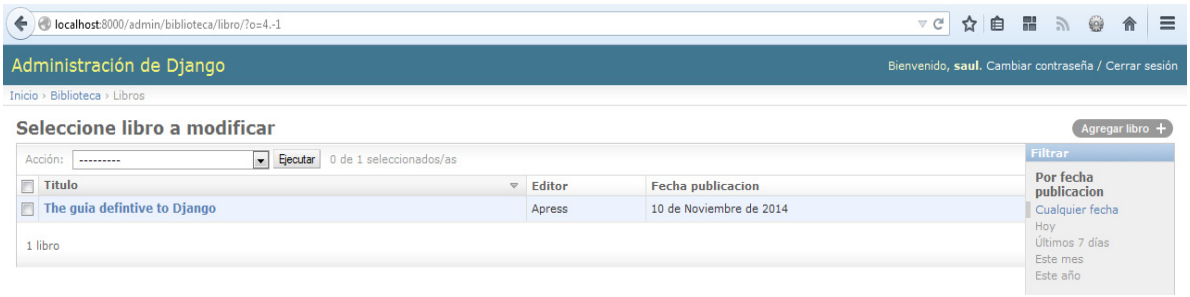


Imagen 6-14. Lista de cambios, después de aplicar algunos filtros.

Aquí vemos porque usamos diferentes tipos de opciones, primero hemos creado una nueva a clase separada de ModelAdmin llamada LibroAdmin. Primero definimos list_display solo para mostrar la lista de cambios de forma más agradable. Luego usamos list_filter, la cual es una tupla de campos que se usa para crear filtros a lo largo de la barra lateral, del lado derecho de la lista de cambios. Para los campos de fechas Django provee algunos atajos para filtrar las listas, tal como “Hoy”, “Últimos 7 días”, “Este mes” y “Este año” –atajos que los desarrolladores de Django han encontrado muy útiles para casos en lo que se necesite filtrar por fechas. La Imagen 6-14 muestra la forma en que lucen.

Otra forma de ofrecer los filtros basados en fechas es usando la opción date_hierarchy así:

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)
    date_hierarchy = 'fecha_publicacion'
```

Con esto en su lugar, la lista de cambios obtiene una barra de navegación desplegable en la parte superior de la lista, como se muestra en la Imagen 6-11. Esta comienza con una lista desplegable de años, de meses y termina con los días de forma individual.

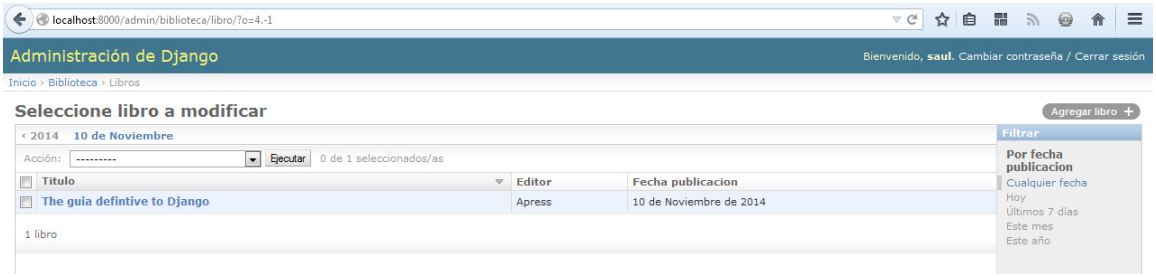


Imagen 6-15. Lista de cambios, después de agregar "date_hierarchy".

Observa que la opcion date_hierarchy toma una *cadena*, no una tupla porque únicamente toma un campo de tipo fecha, el cual ha sido usado para crear la jerarquía.

Finalmente, también podemos cambiar el ordenamiento por omisión de la página de inicio de la lista de cambios, para que siempre sean ordenados en orden descendiente de acuerdo a la fecha de publicación. Por omisión el orden de los objetos en la lista de cambios se da de acuerdo al orden especificado en el modelo con ordering en la class Meta (La cual cubrimos en el capítulo 5) –pero como aun no hemos especificado este valor ordering, el ordenamiento es aun indefinido.

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)
    date_hierarchy = 'fecha_publicacion'
    ordering = ('-fecha_publicacion',)
```

La opción ordering trabaja exactamente como lo hace en los modelos dentro de la clase interna class Meta, excepto que únicamente usa el primer nombre de un campo en la lista. Solo pasa una tupla a la lista de nombres de campos, y agrega un signo (-) al campo para usarlo en orden descendente.

Hasta aquí, hemos cubierto las principales opciones de la lista de cambios. Usando estas opciones, puedes crear interfaces muy poderosas y listas para producir y editar datos, agregando solo algunas pocas líneas de código

Personalizar formularios de edición

Al igual que las listas de cambios que pueden ser hechas a la medida, los formularios para edición pueden personalizarse de muchas maneras.

Primero, personalicemos los campos y la forma en que son ordenados. Por omisión el orden de un campo en una forma o formulario de edición, corresponde al

orden en el que se haya definido en el modelo. Sin embargo podemos cambiar el ordenamiento, usando la opción: fields en una subclase de ModelAdmin.

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)
    date_hierarchy = 'fecha_publicacion'
    ordering = ('-fecha_publicacion',)
    fields = ('titulo', 'autores', 'editor', 'fecha_publicacion')
```

Después de realizar estos cambios, la forma para editar libros, utilizara el orden definido con fields . Es más natural tener autores después del título del libro. Aunque el orden de los campos dependerá de tu flujo de trabajo y de la entrada de los datos que manejes. Cada forma de trabajo es diferente.

Otra cosa útil, la opción fields permite que *excluyas* ciertos campos de un formulario. Solo deja fuera el campo que quieres excluir. Esto puede ser útil si los usuarios administradores en quienes confías únicamente quieren editar una parte de los datos, o si una parte de los campos son cambiados desde fuera digamos, mediante un proceso automático. Por ejemplo en la base de datos biblioteca, podemos ocultar el campo fecha_publicacion para que no sea editable por los usuarios.

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)
    date_hierarchy = 'fecha_publicacion'
    ordering = ('-fecha_publicacion',)
    fields = ('titulo', 'autores', 'editor', 'portada')
```

Como resultado, la forma para editar biblioteca, no ofrece una forma para especificar la fecha de publicación. Esto puede ser útil en algunos casos, digamos por ejemplo que eres un editor que prefiere que sus autores no especifiquen la fecha de publicación (Claro que esto, es solo un ejemplo hipotético.)

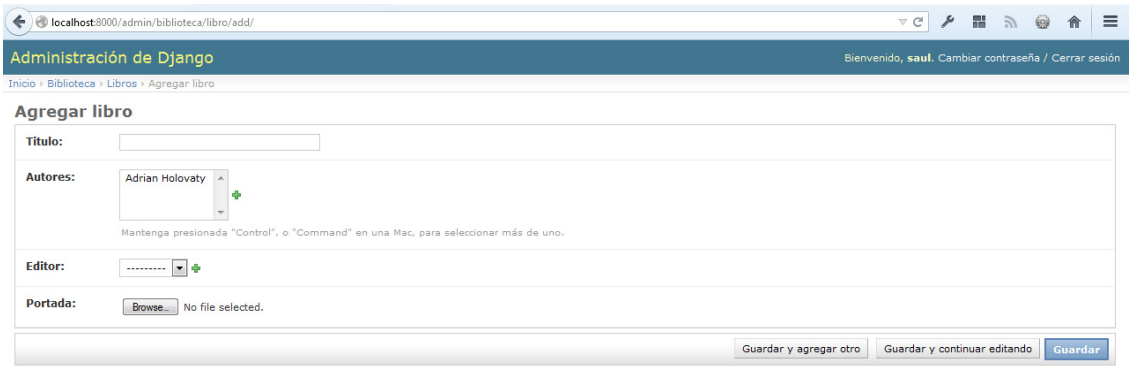


Imagen 6-16. Uso de "field" para mostrar campos

De esta forma cuando un usuario sube el formulario sin el campo fecha_publicacion, para agregar un libro, Django simplemente trata el campo como None – ya que el campo acepta valores nulos, que definimos con la opción: null=True, por lo que puede quedar en blanco.

Otro uso muy común para usar formularios personalizados, se da al usar campos muchos a muchos. Tal como vimos en el formulario para editar libros, en la interfaz administrativa esta cuenta con una caja de selección múltiple, que representa un

campo ManyToManyField, el cual lógicamente usa un widget HTML para la entrada de datos –sin embargo las caja de selección múltiple puede dificultar su uso. Si quieres seleccionar múltiples objetos, tienes que mantener presionada la tecla "control" o "comando" en Mac (el sitio de administración inserta algunos fragmentos de texto en forma de ayuda que explican esto), sin embargo esto se vuelve inmanejable cuando el campo contiene centenares de opciones.

La solución es cambiar la disposición de la interfaz administrativa usando filter_horizontal.

Agreguemos esto a LibroAdmin y veamos lo que ocurre.

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)
    date_hierarchy = 'fecha_publicacion'
    ordering = ('-fecha_publicacion',)
    filter_horizontal = ('autores',)
```

(Si estas siguiendo esto interactivamente, nota que removimos la opción fields para restaurar todos los campos del formulario.)

Recarga el formulario para editar libros y podrás ver que ahora la sección de "Autores" usa una interfaz elegante y un filtro en Java Script que permite explorar a través de las opciones de forma dinámica, lo que permite encontrar autores específicos y permite mover de "Autores disponibles" a la caja "Autores elegidos" y viceversa.

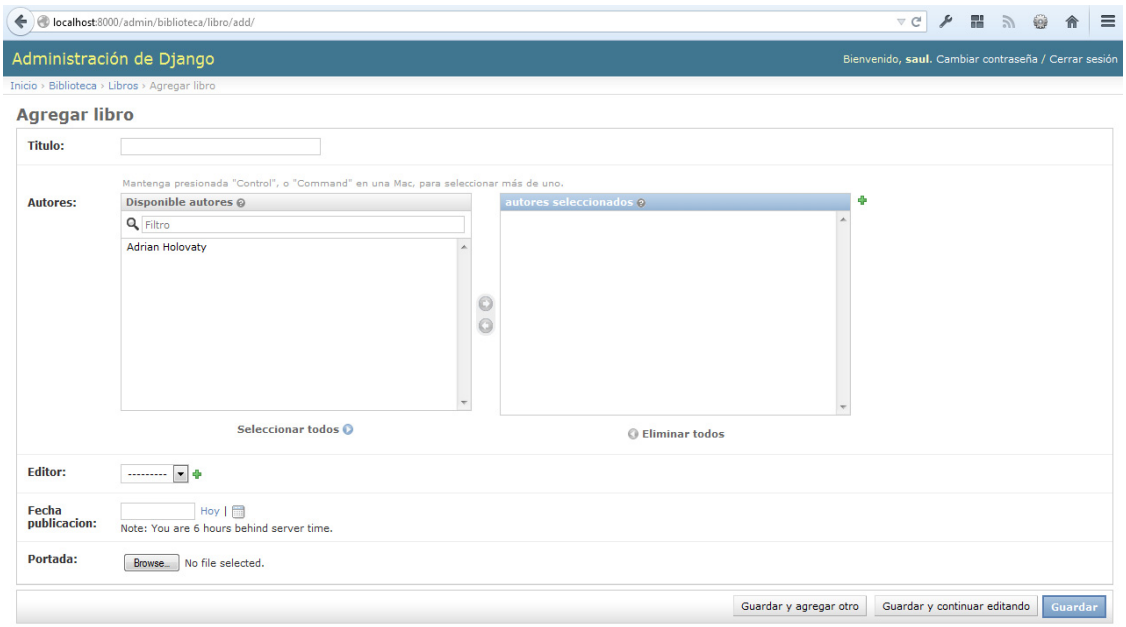


Imagen 6-17. Formulario para editar libros, después de agregar "**filter horizontal**".

Recomendamos usar el filtro horizontal, filter_horizontal para cualquier campo ManyToManyField que contenga más de 10 objetos. Es más sencillo de usar que el widget de selección múltiple. También puedes usar filter_horizontal en campos múltiples –solo especifica cada nombre en una tupla.

La clase ModelAdmin también soporta la opción filtro vertical, **filter_vertical**. La cual trabaja exactamente como filter_horizontal, pero la interfaz Java Script resultante, es una pila que contiene dos cajas verticales en lugar de una. Todo es cuestión de gustos y necesidades.

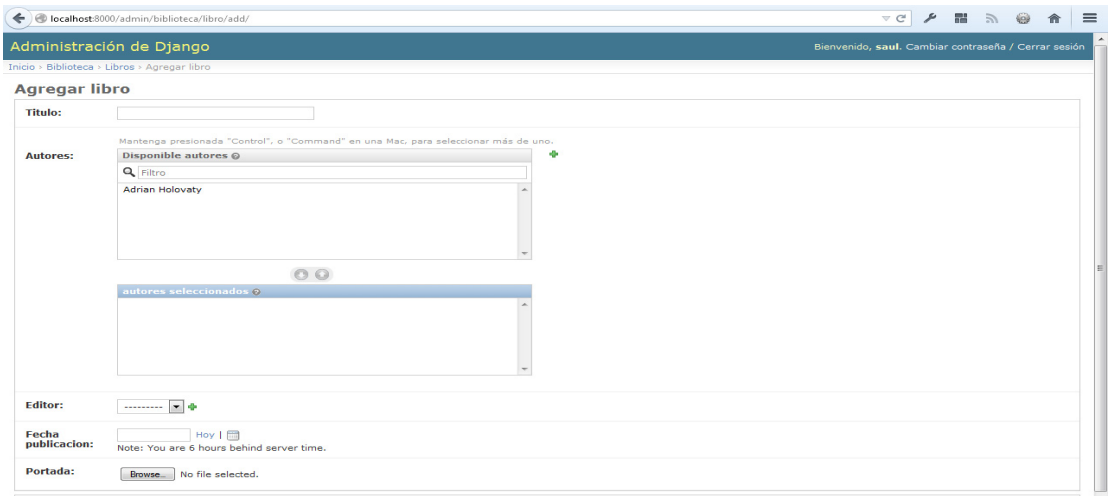


Imagen 6-18. Formulario para editar libros, después de agregar "**filter vertical**".

Los filtros filter_horizontal y filter_vertical únicamente trabajan con campos ManyToManyField no así con campos ForeignKey. Por omisión la interfaz administrativa usa simples cajas select, para mostrar los campos ForeignKey, pero al igual que con los campos ManyToManyField, algunas veces será necesario buscar la forma de no tener que seleccionar todos los objetos relacionados, ya que si nuestra base de datos incluye a millares de editores, la forma para "Agregar libros" puede tardar un buen rato en cargarlos a todos (lo que genera sobrecarga en la base de datos), ya que tendría que cargar a cada editor para mostrar la caja <select>.

La forma de corregir esto es usar una opción llamada raw_id_fields. Colocando en una tupla los nombres de los campos de el ForeignKey, para mostrarlos en la interfaz administrativa, dentro de una simple caja de texto (<input type="text">) en lugar de <select>.

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'editor', 'fecha_publicacion')
    list_filter = ('fecha_publicacion',)
    date_hierarchy = 'fecha_publicacion'
    ordering = ('-fecha_publicacion',)
    filter_horizontal = ('autores',)
    raw_id_fields = ('editor',)
```

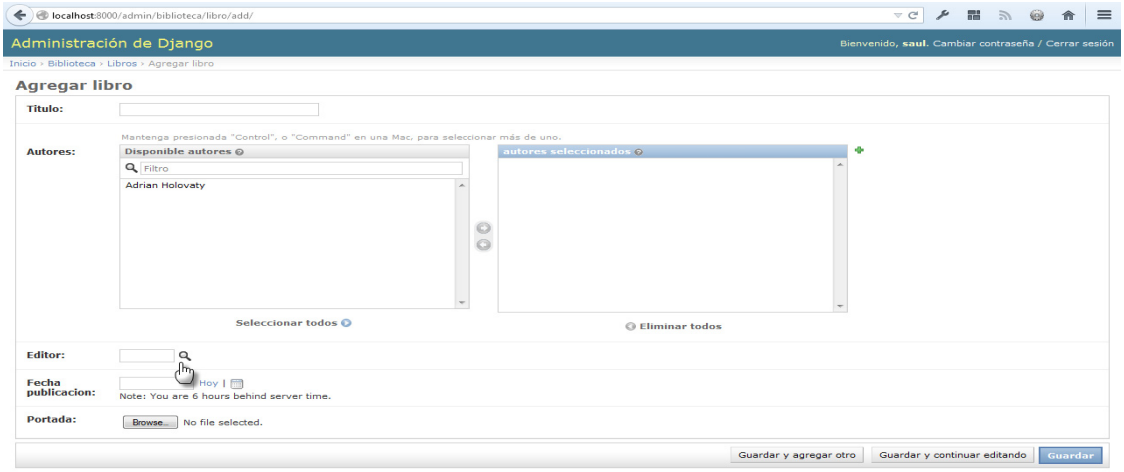


Imagen 6-19. El formulario para editar, después de agregar "raw_id_fields"

¿Y ahora como escribo en esa caja?

La base de datos identifica a cada editor con un ID (una clave primaria). Dado que los seres humanos no memorizamos normalmente identificadores (ID) de las base de datos, existe un icono con forma de lupa que permite, con un simple clic, desplegar una ventana flotante, en la cual se pueden seleccionar al editor para agregarlo a la forma. Por lo que no necesitas escribir directamente en la caja el ID de el objeto.

Personalizar la apariencia de la interfaz de administración

Claramente, tener la frase "Administración de Django" en la cabecera de cada página de administración es ridículo. Es sólo un texto de relleno que es fácil de cambiar, usando el sistema de plantillas de Django. El sitio de administración de Django está propulsado por el mismo

Django, sus interfaces usan el sistema de plantillas propio de Django. (El sistema de plantillas de Django fue presentado en él *capítulo4*.)

Como explicamos en él *Capítulo4*, la configuración de TEMPLATE_DIRS especifica una lista de directorios a verificar cuando se cargan plantillas Django. Para personalizar las plantillas del sitio de administración, simplemente copia el conjunto relevante de plantillas de la distribución Django en uno de los directorios apuntados por TEMPLATE_DIRS.

El sitio de administración muestra "Administración de Django" en la cabecera porque esto es lo que se incluye en la plantilla admin/base_site.html. Por defecto, esta plantilla se encuentra en el directorio de plantillas de administración de Django, django/contrib/admin/templates, que puedes encontrar buscando en tu directorio site-packages de Python, o donde sea que Django fue instalado. Para personalizar esta plantilla base_site.html, copia la original dentro de un subdirectorio llamado admin dentro de cualquier directorio que esté usando TEMPLATE_DIRS. Por ejemplo, si tu directorio TEMPLATE_DIRS incluye "/home/misplantillas", entonces copia django/contrib/admin/templates/admin/base_site.html al directorio /home/misplantillas/admin/base_site.html. No te olvides del subdirectorio admin.

Luego, sólo edita el nuevo archivo admin/base_site.html para reemplazar el texto genérico de Django, por el nombre de tu propio sitio, tal como lo quieres ver.

Nota que cualquier plantilla por defecto de Django Admin puede ser reescrita. Para reescribir una plantilla, haz lo mismo que hicimos con base_site.html: copia esta desde el directorio original a tu directorio personalizado y haz los cambios sobre esta copia.

Puede que te preguntes cómo, si TEMPLATE_DIRS estaba vació al principio, Django encuentra las plantillas por defecto de la interfaz de administración. La respuesta es que, por defecto, Django automáticamente busca plantillas dentro del subdirectorio templates/ de cada paquete de aplicación como alternativa. Mira capítulo 10 para obtener más información sobre cómo funciona esto.

Personalizar la página índice del administrador

En una nota similar, puedes tener la intención de personalizar la apariencia (el *look & feel*) de la página principal del administrador. Por defecto, aquí se muestran todas las aplicaciones, de acuerdo a la configuración que tenga INSTALLED_APPS, ordenados por el nombre de la aplicación. Quizás quieras, por ejemplo, cambiar el orden para hacer más fácil ubicar determinada aplicación que estás buscando. Después de todo, la página inicial es probablemente la más importante de la interfaz de administración, y debería ser fácil utilizarla.

La plantilla para personalizarla es admin/index.html. (Recuerda copiar admin/index.html a tu directorio de plantillas propio como en el ejemplo previo).

Edita el archivo, y verás que usa una etiqueta llamada {% get_admin_app_list as app_list %}. Esta etiqueta devuelve todas las aplicaciones Django instaladas. En vez de usar esta etiqueta, puedes incluir vínculos explícitos a objetos específicos de la manera que creas más conveniente. Si el código explícito en una plantilla no te satisface, puedes ver él *Capítulo 10* para encontrar detalles sobre cómo implementar tus propias etiquetas de plantillas.

Para detalles completos sobre la personalización del sitio de administración de Django, mira él *Capítulo 17*.

Usuarios, Grupos y Permisos

Desde que estás identificado como un superusuario, tienes acceso a crear, editar y eliminar cualquier objeto. Sin embargo, la interfaz de administración tiene un sistema de permisos de usuario que puedes usar para darles a otros usuarios acceso limitado a las partes de la interfaz que ellos necesitan.

Puedes editar estos usuarios y permisos a través de la interfaz de administración, como si fuese cualquier otro objeto. Los vínculos a los modelos Usuarios y Grupos se encuentran en el índice de la página principal junto con todo el resto de los modelos que haz definido.

Los objetos usuario tienen campos estándar: nombre de usuario, contraseña, dirección de correo, y nombre real que puedes esperar, seguidos de un conjunto de campos que definen lo que el usuario tiene permitido hacer en la interfaz de administración. Primero, hay un conjunto de tres opciones seleccionables:

- La opción **Activo** define si el usuario está activo en todo sentido. Si está desactivada, el usuario no tendrá acceso a ninguna URL que requiera identificación.

- La opción **Es staff** indica que el usuario está habilitado a ingresar a la interfaz de administración (por ejemplo, indica que el usuario es considerado un miembro del staff en tu organización). Como el mismo sistema de usuarios puede usarse para controlar el acceso al sitio público (es decir, sitios restringidos no administrativos. Mira él *capítulo 12*.), esta opción diferencia entre usuarios públicos y administradores.

- La opción **es superusuario** da al usuario completo e irrestricto acceso a todos los elementos de la interfaz de administración, y sus permisos regulares son ignorados.

Imagen 6-20. Tipos de permisos.

Los administradores "normales" – esto es, activos, no superusuarios y miembros del staff – tienen accesos que dependen del conjunto de permisos concedidos. Cada objeto editable a través de la interfaz de administración tiene tres permisos: un permiso de *crear*, un permiso de *modificar* y un permiso de *eliminar*. Lógicamente,

asignando permisos a un usuario habilitas que este acceda a realizar la acción que el permiso describe.

■**Nota** El acceso a editar usuarios y permisos también es controlado por el sistema de permisos. Si le das a alguien el permiso de editar usuarios, ¡estará en condiciones de editar sus propios permisos, que probablemente no es lo que querías!
También puedes asígnar usuarios a grupos. Un *grupo* es simplemente un conjunto de permisos a aplicar a todos los usuarios de ese grupo. Los grupos son útiles para otorgar idénticos permisos a un gran número de usuarios.

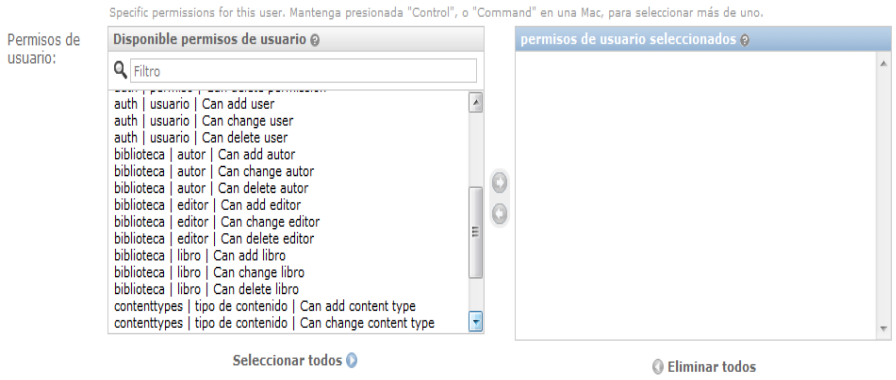


Imagen 6-21. Asignar permisos.

Cuándo y porqué usar la interfaz de administración

Pensamos que la interfaz de administración de Django es bastante espectacular. De hecho, diríamos que es una de sus *killer feautures*, o sea, una de sus características sobresalientes. Sin embargo, a menudo nos preguntan sobre "casos de uso" para la interfaz de administración (¿Cuándo debemos usarlo y por qué?). A lo largo de los años, hemos descubierto algunos patrones donde pensamos que usar la interfaz de administración resulta útil.

Obviamente, es muy útil para modificar datos (se veía venir). Si tenemos cualquier tipo de tarea de introducción de datos, el administrador es difícil de superar. Sospechamos que la gran mayoría de lectores de este libro tiene una horda de tareas de este tipo.

La interfaz de administración de Django brilla especialmente cuando usuarios no técnicos necesitan ser capaces de ingresar datos; ese es el propósito detrás de esta característica, después de todo. En el periódico donde Django fue creado originalmente, el desarrollo de una característica típica online – un reporte especial sobre la calidad del agua del acueducto municipal, supongamos – implicaba algo así:

- El periodista responsable del artículo se reúne con uno de los desarrolladores y discuten sobre la información disponible.
- El desarrollador diseña un modelo basado en esta información y luego abre la interfaz de administración para el periodista.
- Mientras el periodista ingresa datos a Django, el programador puede enfocarse en desarrollar la interfaz accesible públicamente (¡la parte divertida!).

En otras palabras, la razón de ser de la interfaz de administración de Django es facilitar el trabajo simultáneo de productores de contenido y programadores.

Sin embargo, más allá de estas tareas de entrada de datos obvias, encontramos que la interfaz de administración es útil en algunos otros casos:

- *Inspeccionar modelos de datos*: La primer cosa que hacemos cuando hemos definido un nuevo modelo es llamarlo desde la interfaz de administración e ingresar algunos datos de relleno. Esto es usual para encontrar errores de modelado; tener una interfaz gráfica al modelo revela problemas rápidamente.

- *Gestión de datos adquiridos*: Hay una pequeña entrada de datos asociada a un sitio como http://chicagocrime.org, puesto que la mayoría de los datos provienen de una fuente automática. No obstante, cuando surgen problemas con los datos automáticos, es útil poder entrar y editarlos fácilmente.

¿Qué sigue?

Hasta ahora hemos creado algunos modelos y hemos configurando una interfaz administrativa de primera clase para modificar los datos. En él *próximo capítulo*, nos meteremos de lleno en el Desarrollo Web: creando y procesando formularios.

CAPÍTULO 7

Procesamiento de formularios

Los formularios en HTML, son la columna vertebral de los sitios Web interactivos, os encontramos como simples caja de búsquedas como los que usa Google o como los inconfundibles formularios para subir comentarios siempre presentes en la mayoría de blogs, pero también existen complejas interfaces que permiten la entradas de datos interactivamente y de forma personalizada, y es que los formularios son una necesidad básica en la mayoría de aplicaciones Web modernas que necesitan recopilar datos a través de internet.

Este capítulo cubre la manera de usar Django para recopilar datos a través de formularios, validarlos y hacer algo útil con ellos. A lo largo de este capítulo nos enfocaremos en conocer los objetos HttpRequest y los objetos Form

Comenzaremos creando un simple formulario de búsquedas "a mano", observando cómo manejar los datos suministrados al navegador. Y a partir de ahí, pasaremos al uso del *framework* de formularios que viene incluido en Django.

Obteniendo datos de los objetos "Request"

Introducimos los objetos HttpRequest en el *capítulo 3*, cuando cubrimos las funciones vista, pero no tuvimos mucho que decir acerca de ellos en aquel momento.

Recuerdas que cada función de vista toma un objeto HttpRequest como primer parámetro, tal como en la vista hola() que construimos:

```
from django.http import HttpResponse

def hola(request):
    return HttpResponse("Hola mundo")
```

Los objetos HttpRequest, tal como la variable request, contienen un número de atributos y métodos interesantes con los cuales deberías familiarizarte, a fin de saber cómo utilizarlos lo mejor posible. Puedes utilizar estos atributos para conseguir información acerca de las peticiones que recibes (por ejemplo: el usuario o el navegador que está cargando la pagina en tu sitio creado con Django), al mismo tiempo que la función vista es ejecutada.

INFORMACIÓN ACERCA DE LAS URL

Los objetos HttpRequest contienen algunas piezas de información acerca de la URL requerida.

La siguiente tabla muestra los métodos o atributos de las peticiones request:

Atributos o Métodos	Descripción	Ejemplo
request.path	La ruta completa, no incluye el dominio pero incluye, la barra inclinada.	"/hola/"
request.get_host()	El host (ejemplo: tu "dominio," en lenguaje común).	"127.0.0.1:8000" o "www.example.com"
request.get_full_path()	La ruta (path), mas una cadena de consulta (si está disponible).	"/hola/?print=true"
request.is_secure()	True si la petición fue hecha vía HTTPS. Si no, False.	True o False

Tabla 7.1 Información acerca de las URL

Siempre usa estos atributos/métodos en lugar de incrustar las URLs en tus vistas, esto hace más flexible tu código y más fácil de usar en otros lugares u otras aplicaciones. Un simple ejemplo:

```
# ¡Mal! ☠
def vista_actual_url(request):
    return HttpResponse("Bienvenido a mi pagina en /pagina_actual/")

# Bien ☺
def vista_actual_url(request):
    return HttpResponse("bienvenido a mi pagina en %s" % request.path)
```

MÁS INFORMACIÓN ACERCA DE LAS PETICIONES O REQUEST

request.META es un diccionario Python, que contiene todas las cabeceras HTTP disponibles para la petición dada –Incluyendo la dirección IP y el agente– Generalmente el nombre y la versión del navegador Web. Observa que la lista completa de cabeceras disponibles depende de las cabeceras que el usuario envía y las cabeceras que el servidor Web seleccione. Algunas de las claves más comunes están disponibles como diccionarios y son las siguientes:

- **HTTP_REFERER:** La respectiva URL, a la que se hace referencia. (Observa la forma en que se escribe REFERER.)
- **HTTP_USER_AGENT:** El navegador del usuario en forma de cadena, cualquiera que sea. Esta es algo si: "Mozilla/5.0 (X11; U; Linux i686; fr-FR; rv: 1.8.1.17) Gecko/20080829 Firefox/2.0.0.17".
- **REMOTE_ADDR:** La dirección IP del cliente, por ejemplo: "12.345.67.89". (Si la petición ha pasado a través de un proxi, entonces puede retornar una lista separada por comas, conteniendo las direcciones IP, por ejemplo: "12.345.67.89, 23.456.78.90".)

Observa que request.META es básicamente un diccionario Python, que contiene todas las cabeceras HTTP disponibles, las que dependen del navegador y del servidor que se esté usando en ese determinado momento, por lo que obtendrás una excepción KeyError si intentas acceder a una clave que no existe. (Además las cabeceras HTTP son datos *externos* – que son subidos por los navegadores de los usuarios, ***por lo que no deberías confiar en ellos***, así que siempre diseña tus aplicaciones para que fallen intencionalmente, si una cabecera en particular está

vacía o no existe.) Aprende a usar las clausulas try/except o el método get() para manejar los casos en que alguna de estas claves estén indefinidas, para evitar errores:

```
# ¡Mal! ☠
def mostrar_navegador(request):
    ua = request.META['HTTP_USER_AGENT'] # ¡Podría lanzar: KeyError!
    return HttpResponse("Tu navegador es %s" % ua)

# Bien (Versión 1)
def mostrar_navegador(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Tu navegador es %s" % ua)

# Bien (Versión 2)
def mostrar_navegador2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Tu navegador es %s" % ua)
```

Te animamos a escribir pequeñas vistas como estas, que muestren todos los datos request.META, que puedas conseguir para familiarizarte con los conceptos. La vista puede empezar así:

```
def atributos_meta(request):
    valor = request.META.items()
    valor.sort()
    html = []
    for k, v in valor:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
   return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

Como ejercicio trata de convertir la vista anterior, para que use el sistema de plantillas en lugar de incrustar el código HTML en la vista. También trata de agregar request.path y los otros métodos HttpRequest que vimos en la sección anterior.

INFORMACIÓN ACERCA DE LOS DATOS RECIBIDOS

Más allá de los metadatos básicos obtenidos de las peticiones Web, los objetos HttpRequest poseen dos atributos mas, que contienen información recibida de los usuarios: request.GET y request.POST. Ambos atributos son como objetos tipo diccionarios que permiten el acceso a datos GET y POST

¿OBJETOS COMO DICCIONARIOS?

Cuando decimos que request.GET y request.POST son como objetos tipo diccionario (“dictionary-like”), lo que tratamos de decirte es que se comportan como diccionarios estándar de Python, pero técnicamente en el fondo no son diccionarios. Por ejemplo request.GET y request.POST contienen ambos, métodos como get(), keys() y values(), por lo que puede iterarse sobre sus claves usando un bucle como: for key in request.GET.

¿Entonces porque la distinción? bueno, porque ambos request.GET y request.POST contienen métodos adicionales, que los diccionarios normales no, llegaremos a eso dentro de poco.

Puede ser que encuentres el termino similar a “file-like objects” – Objetos Python que contienen algunos métodos básicos como read(), lo que le permite actuar a un determinado archivo como un “objeto”.

Los datos POST generalmente son recibidos de formularios (<form>) HTML, mientras que los datos GET son enviados a los formularios (<form>) o mediante una cadena de consulta a una página URL.

Tu primer formulario creado con Django

Continuando con el ejemplo en curso sobre: libros, autores y editores, vamos a crear un formulario, mediante una vista muy simple que permita a los usuarios buscar libros en la base de datos mediante el titulo.

Generalmente, se necesitan dos partes para desarrollar un formulario: la interfaz de usuario en HTML y la vista que procesa los datos obtenidos o subidos por los usuarios. La primera parte es sencilla; solo necesitamos crear una vista que muestre el formulario de búsqueda:

biblioteca/views.py

```
from django.shortcuts import render

def formulario_buscar(request):
    return render(request, 'formulario_buscar.html')
```

Tal como aprendimos en el *capítulo 3*, la vista puede estar en cualquier lugar de la ruta de búsqueda de Python. Pero por convención este debe de ir en una vista, podemos colocarla en biblioteca/views.py.

Acompañada de una plantilla llamada formulario_buscar.html, que debe ubicarse en un directorio llamado templates (de forma predeterminada Django busca las plantillas en un directorio interno llamado template en cada aplicación registrada), dentro del directorio de la aplicación biblioteca, en el mismo nivel que el directorio migrations:

biblioteca/templates/formulario_buscar.html

```
<html>
<head>
  <title>Buscar</title>
</head>
<body>
  <form action="/buscar/" method="get">
    <input type="text" name="q">
    <input type="submit" value="Buscar">
  </form>
</body>
</html>
```

Y para acceder al formulario, necesitamos el patrón para la URL:

misitio/urls.py

```
from biblioteca import views

urlpatterns = [
  # ...
  url(r'^formulario-buscar/$', views.formulario_buscar),
]
```

(Observa que hemos importando el modulo views directamente, en lugar de hacer algo como esto: from biblioteca.views import formulario_buscar, porque lo anterior es más elegante y entendible.

Veremos la forma de aprovechar este tipo de importaciones en más detalle, en el *capítulo 8*.

Ahora, ejecuta el comando python manage.py runserver o recarga el servidor y visita la pagina: http://127.0.0.1:8000/formulario-buscar/, donde veras una interfaz de búsqueda, bastante simple, construida mediante un sencillo formulario:

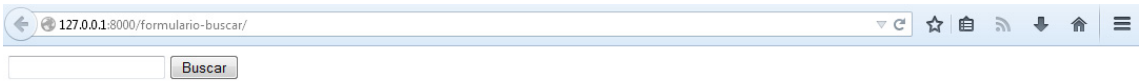


Figura 7-1. Ejemplo de un formulario de búsquedas.

Trata de subir el formulario, y solo conseguirás un error 404. El formulario que apunta a la URL /buscar/, aun no ha sido implementado. Reparemos eso con una segunda función de vista y su respectiva URLconf:

misitio/urls.py

```
urlpatterns = [
    # ...
    url(r'^formulario-buscar/$', views.formulario_buscar),
    url(r'^buscar/$', views.buscar),
    # ...
]
```

biblioteca/views.py

```
from django.http import HttpResponse

def buscar(request):

    if 'q' in request.GET and request.GET['q']:
        mensaje = 'Estas buscando: %r' % request.GET['q']
    else:
        mensaje = 'Haz subido un formulario vacio.'
    return HttpResponse(mensaje)
```

Por el momento, esto exhibe meramente el término de búsqueda del usuario, así que podemos estar seguros de una cosa, los datos están siendo enviados a Django correctamente, y puede darnos una ligera percepción sobre la forma en que el término de búsqueda atraviesa el sistema.

En resumen:

- El formulario HTML (<form>) define una variable q. Cuando esta es subida el valor de q es enviado mediante el método GET a la URL /buscar/.

- La vista de Django maneja la URL /buscar/ y tiene acceso al valor de q en la petición request.GET.

Una cosa que es importante precisar aquí, es que explícitamente verificamos que 'q' exista en request.GET. Como precisamos en la sección anterior con las peticiones request.META, no deberías confiar en nada que sea subido por los usuarios o incluso asume que no subieron nada en primer lugar. Si no agregas esta verificación, los formularios vacios lanzaran un error del tipo KeyError en la vista:

```
# MAL! ☠
def buscar_no_hagas_esto(request):
    #¡Las siguientes lineas lanzan un error "KeyError" si no se envía 'q'!
    mensaje = 'Estas buscando: %r' % request.GET['q']
    return HttpResponse(mensaje)
```

PARÁMETROS DE CADENAS DE CONSULTA

Porque los datos GET se pasan en cadenas de consultas (por ejemplo: /buscar/?q=django) puedes usar request.GET para acceder a las variables de las cadenas de consulta. En el capítulo 3, "Introducción a los patrones URLconfs de Django", comparamos las URL bonitas contra las tradicionales URLs de PHP/Java tal como /tiempo/mas?horas=3 y dijimos que te mostraríamos como hacerlo más adelante en el capítulo 7. Ahora ya sabes cómo acceder a cadenas de parámetros en las vistas (tal como el ejemplo horas=3) –Solo usa request.GET.

Los datos POST trabajan de la misma forma que lo datos GET – solo usa request.POST en lugar de request.GET.

¿Entonces cual es la diferencia entre GET y POST? Usa GET cuando el acto de subir un formulario solo sea para "pedir" datos. En cambio usa POST siempre que el acto de subir el formulario tenga efectos secundarios – *cambiando* datos, enviando un e-mail o algo que vaya más allá de simplemente *mostrar* datos. En el ejemplo de búsquedas del ejemplo, estamos usando GET porque la consulta no cambia ningún dato en nuestro servidor.

(Consulta http://www.w3.org/2001/tag/doc/whenToUseGet.html si quieres aprender más sobre las peticiones GET and POST.)

Ahora que hemos verificado que el método request.GET es pasado apropiadamente, anclemos la consulta de búsquedas a la base de datos. (Otra vez en views.py y rescribamos la función buscar):

biblioteca/views.py

```
from django.http import HttpResponse
from django.shortcuts import render

from biblioteca.models import Libro

def buscar(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        libros = Libro.objects.filter(titulo__icontains=q)
        return render(request, 'resultados.html',  {'libros': libros, 'query': q})
    else:
        return HttpResponse('Por favor introduce un termino de búsqueda.')
```

Un par de notas sobre lo que hicimos:

- Aparte de checar que 'q' exista en request.GET, nos aseguramos que request.GET['q'] no sea una cadena vacía antes de pasarle la consulta a la base de datos.

- Estamos usando Libro.objects.filter(titulo__icontains=q) para consultar en la tabla libros, todos los libros que incluyan en el titulo los datos proporcionados en la consulta, icontains es un tipo de búsqueda (como explicamos en el capítulo 5 y el apéndice B) en la que no se distinguen mayúsculas de minúsculas (*case-insensitive*), y que internamente usa el operador LIKE de SQL en la base de datos. La declaración puede ser traducida como "Obtener todos los libros que contengan estas palabras"

  Esta es una forma muy simple para buscar libros. No recomendamos usar una simple consulta icontains en bases de datos muy grandes en producción ya que esto puede ser muy lento. (En el mundo real, es mejor usar un sistema de búsqueda personalizado de cierto tipo. Busca en la Web *proyectos libres de sistemas de búsquedas de texto* para que te des una idea de las posibilidades.)

Pasamos libros, como una lista de objetos Libro a la plantilla. El código de la plantilla resultados.html debe incluir algo como esto:

biblioteca/templates/resultados.html

```
<p>Estas buscado: <strong>{{ query }}</strong></p>

{% if libros %}
    <p>Libros encontrados: {{ libros|length }} libro{{ libros|pluralize }}.</p>
    <ul>
        {% for libros in libros %}
            <li>{{ libros.titulo }}</li>
        {% endfor %}
    </ul>
{% else %}
    <p>Ningún libro coincide con el criterio de búsqueda.</p>
{% endif %}
```

Observa que estamos usando el filtro de plantillas pluralize, el cual apropiadamente agrega en la salida el plural (la "s"), basado en el número de libros encontrados en la base de datos.

Mejorando la forma de manejar un formulario

En los párrafos anteriores te mostramos la forma más simple en la que podría trabajar un formulario. Ahora te mostraremos algunos problemas que pueden surgir y la manera de solucionarlos.

Primero, la forma en que la vista buscar() maneja las consultas vacías es pobre – solo mostramos un mensaje "Por favor introduce un termino de búsqueda." lo que requiere que el usuario, tenga que dar clic de nuevo en el botón para regresar su navegador a la página de búsquedas. Esto es horrible y poco profesional, si implementas algo así, tus privilegios Django serán revocados.

Sería mucho mejor volver a mostrar el formulario con los errores resaltados, para que el usuario pueda intentar nuevamente rellenarlo. La forma más fácil de hacer esto, es renderizando la plantilla otra vez, usando una clausula else; así:

biblioteca/views.py

```
from django.http import HttpResponse
from django.shortcuts import render

from biblioteca.models import Libro

def formulario_buscar(request):
    return render(request, 'formulario_buscar.html')

def buscar(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        libros = Libro.objects.filter(titulo__icontains=q)
        return render(request, 'resultados.html',{'libros': libros, 'query': q})
    else:
        return render(request, 'formulario_buscar.html', {'error': True})
```

(Observa que hemos incluido la vista formulario_buscar(), para que puedas observar ambas vistas en un solo lugar.)

También hemos mejorado la vista buscar() para renderizar la plantilla resultados.html otra vez, si la consulta está vacía. Ya que necesitamos mostrar los mensajes de errores en la plantilla, hemos pasado la variable error a la plantilla. Ahora edita formulario_buscar.html y checa la variable error que hemos agregado:

biblioteca/templates/formulario_buscar.html

```
<html>
<head>
    <title>Buscar</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Por favor introduce un término de búsqueda.</p>
    {% endif %}
    <form action="/buscar/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Buscar">
    </form>
</body>
</html>
```

Con este cambio en su lugar, tenemos una mejor aplicación, pero ahora nos preguntamos ¿Es realmente necesaria una vista dedicada como formulario_buscar(), para realizar este proceso? Tal y como están las cosas, una petición a una URL /buscar/ (http://127.0.0.1:8000/buscar/, sin parámetros GET) mostrara un formulario vacio (mostrándonos el mensaje de error). Podemos remover la vista formulario_buscar(), junto con los patrones asociados a la URL, así como necesitamos cambiar la vista buscar() para que esconda el mensaje de error cuando alguien visite /buscar/ sin parámetros GET:

biblioteca/views.py

```
def buscar(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            libros = Libro.objects.filter(titulo__icontains=q)
            return render(request, 'resultados.html', {'libros': libros, 'query': q})

    return render(request, 'formulario_buscar.html', {'error': error})
```

Con esta vista actualizada, cuando un usuario visita /buscar/ sin parámetros GET, el formulario de búsqueda no mostrara los mensajes de error. Si un usuario trata de subir un formulario con un valor vacio para 'q', el formulario de búsqueda mostrara el mensaje de error en letras rojas. Y finalmente si un usuario sube un formulario con una cadena de consulta – es decir con valores no vacios para 'q', se mostraran los resultados de la búsqueda.

Podemos realizar una mejora final para nuestra aplicación, removiendo un poco de redundancia. Ahora que hemos comenzado a refinar las dos vistas y las URLs en una sola llamada /buscar/, que manejara el despliegue y el formulario de búsquedas y también mostrara los resultados, por lo que el formulario en HTML en la plantilla formulario_buscar.html no necesita que se incruste el código en la URL, en lugar de usar esto:

```
<form action="/buscar/" method="get">
```

Podemos cambiarlo por esto:

```
<form action="" method="get">
```

action="" significa “Envía el formulario con la misma URL a la pagina actual”. Con este cambio realizado, no tienes que recordar cambiar *action* cada vez que necesites enlazar la vista buscar() a otra URL.

Validación simple

Nuestro ejemplo de búsquedas es razonablemente simple, específicamente hablando en términos de validación de datos, ya que solamente nos aseguramos que las consultas en las búsquedas no estén vacías. Muchos formularios en HTML incluyen niveles de validación más complejos, que van más allá de solo asegurarse que los valores no estén vacios. Todos hemos visto mensajes de error en sitios Web como estos.

- “Por favor introduce una dirección de correo electrónica valida”. ‘foo' no es una dirección de correo electrónica.
- “Por favor introduce un código postal valido de 5 dígitos”. ‘123 no es un código postal valido'
- “Por favor introduce una fecha valida en el formato YYYY-MM-DD.”
- “Por favor introduce una contraseña que contenga al menos 8 caracteres y que contenga al menos un número”.

Una nota sobre validación usando Java Script

Este tema va más allá del alcance de este libro, pero puedes usar Java Script para validar datos del lado del cliente, directamente en el navegador. Pero ten cuidado: incluso si haces esto, también *debes* de validar los datos del lado del servidor. Algunas personas pueden tener Java Script desactivado y algunos usuarios maliciosos pueden subir en crudo, datos no validos directamente al manejador de formularios, para ver que daños pueden causar sus travesuras.

No hay nada que hacer en estos casos, con excepción de *siempre* validar los datos enviados por los usuarios de el lado del servidor (Por ejemplo en las vista de Django). Debes pensar en usar la validación en Java Script como una característica extra de usabilidad, no como la única manera de validación.

Bien, ajustemos la vista buscar() para que valide términos de búsqueda que contengan como máximo 20 caracteres o menos. (Para efectos del ejemplo, digamos que los términos largos pueden hacer las consultas muy lentas.) ¿Cómo podemos hacer eso? La cosa más simple posible seria pensar en incrustar directamente la lógica en la vista, más o menos así:

biblioteca/views.py

```
def buscar(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        elif len(q) > 20:
            error = True
        else:
            libros = Libro.objects.filter(titulo__icontains=q)
            return render(request, 'resultados.html',{'libros': libros, 'query': q})
    return render(request, 'formulario_buscar.html',{'error': error})
```

Ahora, si tratamos de enviar una consulta que contenga más de 20 caracteres de longitud, no nos permitirá buscar, solo obtendremos un mensaje de error. Pero el mensaje de error actual en formulario_buscar.html dice: "Por favor introduce un término de búsqueda." Por lo que tendremos que cambiarlo para ser más precisos:

biblioteca/templates/formulario_buscar.html

```
<html>
<head>
    <title>Buscar</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Por favor introduce un termino de búsqueda menor a
        20 caracteres. </p>
    {% endif %}
    <form action="/buscar/" method="get">
        <input type="text" name="q">
```

```
        <input type="submit" value="Buscar">
    </form>
</body>
</html>
```

Hay algo muy raro en esto. El único mensaje de error es potencialmente confuso. ¿Por qué el mensaje para el envió de un formulario vacio menciona un límite de 20 caracteres? Los mensajes de error deben ser específicos, no deben dar lugar a ambigüedades y no deben ser confusos.

El problema está en el hecho de que estamos usando un simple valor booleano para manejar el error, mientras que deberíamos usar una *lista* de cadenas para mostrar el mensaje de error. Esta es la forma en que podemos arreglarlo:

biblioteca/views.py

```
def buscar(request):
    errors = []
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            errors.append('Por favor introduce un término de búsqueda.')
        elif len(q) > 20:
            errors.append('Por favor introduce un término de búsqueda menor a
                20 caracteres.')
        else:
            libros = Libro.objects.filter(titulo__icontains=q)
            return render(request, 'resultados.html',{'libros': libros, 'query': q})

    return render(request, 'formulario_buscar.html', {'errors': errors})
```

Ahora, necesitamos hacer algunos pequeños cambios a la plantilla formulario_buscar.html para que refleje ahora la forma en que estamos pasando la lista de *errors* en lugar de un valor booleano error.

biblioteca/templates/formulario_buscar.html

```
<html>
<head>
    <title>Buscar</title>
</head>
<body>
    {% if errors %}
        <ul>
            {% for error in errors %}
            <li style="color: red;">{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <form action="/buscar/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Buscar">
    </form>
</body>
</html>
```

De esta forma podemos validar de forma simple las cadenas de consultas:

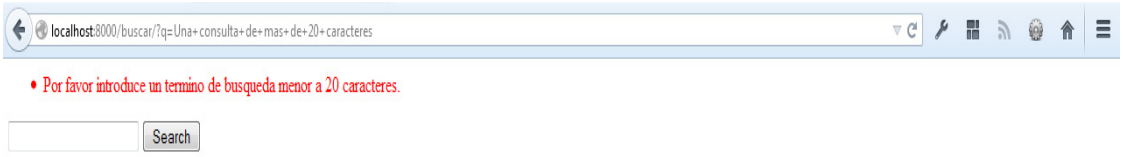


Figura 7-2. Ejemplo de un formulario para validar cadenas de datos.

Construir un formulario para contactos

A pesar de que buscamos mejorar el formulario de búsquedas en el ejemplo pasado, varias veces y lo mejoramos de forma elegante, sigue siendo fundamentalmente simple; ya que contiene únicamente un campo 'q'. Debido a que era tan simple, no utilizamos la librería de formularios de Django, para tratar con ello. Pero los formularios más complejos, necesitan un tratamiento más complicado – y ahora desarrollaremos algo más complicado: un formulario para un sitio de contactos.

El formulario de contactos permitirá a los visitantes del sitio enviar un poco de retroalimentación junto con una dirección e-mail opcional. Después de que el formulario sea enviado y los datos validados, automáticamente enviara un mensaje vía e-email al personal del sitio Web.

Empezamos con la plantilla, formulario-contactos.html.

contactos/templates/formulario-contactos

```
<html>
<head>
   <title>Contactanos</title>
</head>
<body>
   <h1>Contactanos</h1>
{% if errors %}
   <ul>
   {% for error in errors %}
      <li>{{ error }}</li>
   {% endfor %}
   </ul>
{% endif %}
<form action="/contactos/" method="post">{% csrf_token %}
   <p>Asunto: <input type="text" name="asunto"></p>
   <p>E-mail (opcional): <input type="text" name="email"></p>
   <p>Mensaje: <textarea name="mensaje" rows="10" cols="50"></textarea></p>
   <input type="submit" value="Enviar">
</form>
</body>
</html>
```

Definimos tres campos en la plantilla: el asunto, la dirección de correo electrónico y el mensaje. El segundo es opcional, pero los otros dos campos son obligatorios. Observa que estamos usando el método="post" en lugar de el método="get" porque al enviar el formulario, este tiene efectos secundarios –envía un e-mail. También observa que copiamos la forma de mostrar los errores de el código, de la anterior plantilla formulario_buscar.html.

Si continuamos siguiendo el camino, que establecimos al crear la vista buscar() de la sección anterior, escribimos la URLconf y construimos una versión preliminar de la vista contactos() así:

misitio/urls.py

```
from contactos.views  import contactos

urlpatterns = [
    # ...
   url(r'^contactos/$', contactos),
   # ...
]
```

contactos/views.py

```
from django.core.mail import send_mail
from django.http import HttpResponseRedirect
from django.shortcuts import render

def contactos(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('asunto', ''):
            errors.append('Por favor introduce el asunto.')
        if not request.POST.get('mensaje', ''):
            errors.append('Por favor introduce un mensaje.')
        if request.POST.get('email') and '@' not in request.POST['email']:
            errors.append('Por favor introduce una direccion de e-mail valida.')
        if not errors:
            send_mail( request.POST['asunto'],    request.POST['mensaje'],
                request.POST.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],  )

            return HttpResponseRedirect('/contactos/gracias/')
    return render(request, 'formulario-contactos.html', {'errors': errors})
```

(Si continuas siguiendo los ejemplos, tal vez te preguntes, ¿Si debes poner la vista en el archivo biblioteca/views.py. Aunque este no tenga nada que ver con la aplicación biblioteca, o debes de ponerla en otro lugar? Esta decisión es tuya; a Django no le importa, con tal de que la vista apunte a la URLconf. Aunque personalmente deberías crear un directorio separado, algo así como: contactos, en el mismo nivel en el que esta la aplicación biblioteca, en el árbol de directorios, conteniendo un archivo vacio __init__.py y una vista views.py.

Algunas novedades que pasan aquí:

- Estamos comprobando que request.method sea 'POST'. Esto únicamente será verdadero en los casos en que se envié el formulario y no en el caso de que alguien simplemente mira el formulario de contactos (En este último caso request.method será fijado como 'GET', porque los navegadores Web normalmente exploran usando GET, no POST.) Esto hace más agradable aislar "El formulario para mostrar" de los casos en que se necesite presentar el "Procesamiento de formularios".

- En lugar de usar request.GET, estamos usando request.POST para acceder a los datos del formulario de envió. Esto es necesario porque el formulario

HTML en formulario-contactos.html usa method="post". Si la vista es accedida vía POST, en lugar de request.GET estará vacía.

- Esta vez tenemos *dos* campos requeridos, asunto y mensaje, así que tenemos que validar ambos. Observa que utilizamos request.POST.get() y proveemos una cadena en blanco como el valor por omisión para el tercer campo; esta es una manera agradable y corta de manejar ambos casos, para evitar que falten claves y se pierdan datos.

- Aunque el campo de email no es requerido, todavía podemos validarlo si nos lo envían. Nuestro algoritmo de validación es frágil – solo comprobamos que la cadena contenga un carácter @. En el mundo real, necesitamos una validación más robusta (y Django nos la proveerá, en breve te mostraremos como.)

- Estamos usando la funcion django.core.mail.send_mail para enviar un e-mail. Esta función tiene cuatro argumentos obligatorios: el asunto y el cuerpo del mensaje, la dirección del emisor, y una lista de direcciones del destino. send_mail es un wrapper en torno a la clase EmailMessage de Django, la cual provee características avanzadas tales como archivos adjuntos, mensajes múltiples y un control completo sobre los encabezados del mensaje.

- Ten en cuenta que para usar el envió de e-mail usando send_mail(), tu servidor debe de estar configurado para enviar emails, y Django debe de informar al servidor sobre la salida de e-mails. Más adelante configuraremos un servidor de correo que te permitirá enviar emails en el proceso de desarrollo.

- Después de que el e-mail es enviado con "éxito", redirige al usuario a otra página retornando un objeto HttpResponseRedirect. Te dejaremos la implementación de la pagina "enviado con éxito" (ya que es una simple vista/URLconf/plantilla) pero es necesario explicar que debes usar un redirecionamiento para dirigir al usuario a otro lugar, en vez de por ejemplo, simplemente llamar a render() con una plantilla allí mismo.

Esta es la razón: Si un usuario selecciona *actualizar* sobre una página que muestra una consulta POST, la consulta se repetirá. Esto probablemente lleve a un comportamiento no deseado, por ejemplo, que el registro se agregue dos veces a la base de datos. Redirigir luego del POST es un patrón útil que puede ayudarte a prevenir este escenario. Así que luego de que se haya procesado el POST con éxito, *siempre* redirige al usuario a otra página en lugar de retornar el HTML directamente. Esta es una buena práctica de desarrollo Web

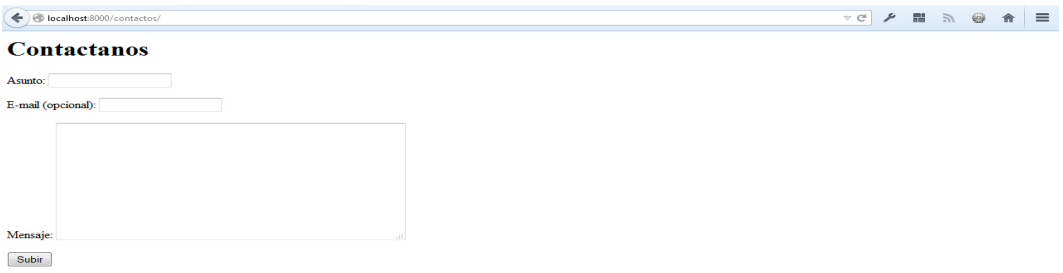


Figura 7-3. Ejemplo de un formulario para contactos.

La vista trabaja, pero algunas funciones de validación son “repetitivas” Imagina procesar un formulario con una docena de campos, ¿Realmente quieres escribir todas esas declaraciones if?
Otro problema es si el usuario ha cometido algún error, *el formulario debería volver a mostrarse*, junto a los mensajes de error resaltados. Los campos deberían rellenarse con los datos previamente suministrados, para evitarle al usuario tener que volver a tipear todo nuevamente. (El formulario debería volver a mostrarse una y otra vez, hasta que todos los campos se hayan rellenado correctamente.) Podemos devolver *manualmente* los datos POST a la plantilla, pero tendríamos que corregir los campos en HTML para insertar el valor apropiado en el lugar correcto.

Finalmente esta es la vista y la plantilla final:

contactos/views.py

```
def contactos(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('asunto', ''):
            errors.append('Por favor introduce el asunto.')
        if not request.POST.get('mensaje', ''):
            errors.append('Por favor introduce un mensaje.')
        if request.POST.get('email') and '@' not in request.POST['email']:
            errors.append('Por favor introduce una direccion de e-mail valida.')
        if not errors:
            send_mail(
                request.POST['asunto'],
                request.POST['mensaje'],
                request.POST.get('email', 'noreply@example.com'),
                    ['siteowner@example.com'], )

            return HttpResponseRedirect('/contactos/gracias/')
    return render(request, 'formulario-contactos.html', {'errors': errors,
        'asunto': request.POST.get('asunto', ''),
        'mensaje': request.POST.get('mensaje', ''),
        'email': request.POST.get('email', ''),
     })
```

contactos/templates/formulario-contactos.html

```
<html>
<head>
    <title>Contactanos</title>
</head>
<body>
    <h1>Contactanos</h1>
{% if errors %}
<ul>
    {% for error in errors %}
        <li>{{ error }}</li>
    {% endfor %}
</ul>
{% endif %}

<form action="/contactos/" method="post">{% csrf_token %}
```

```
    <p>Asunto: <input type="text" name="asunto" value="{{ asunto }}"></p>
    <p>E-mail (opcional): <input type="text" name="email" value="{{ email }}"></p>
    <p>Mensaje: <textarea name="mensaje" rows="10" cols="50"
    value="{{ mensaje }}"></textarea></p>  <input type="submit" value="Enviar">
</form>
</body>
</html>
```

Como seguramente te habrás dado cuenta, el proceso de validación de datos, no es una tarea sencilla, ya que introduce una buena cantidad de oportunidades para cometer errores humanos.

Esperamos que comiences a ver la oportunidad que ofrecen algunas bibliotecas de alto nivel de Django, que manejan los trabajos relacionados con la validación, por ti.

Tu primer formulario usando clases

Django posee una librería llamada django.forms, que maneja muchos tipos de temas que hemos explorado en este capítulo –Formularios para validar y mostrar HTML. Sumerjámonos dentro de nuestra aplicación para formularios de contactos, usando ahora los formularios del framework Django.

La librería "forms" de Django: A través de la comunidad de Django, puedes haber leído o escuchado alguna conversación sobre algo llamado django.newforms. Cuando la gente pregunta sobre django.newforms, ellos se refieren ahora a django.forms – la librería que cubriremos en este capítulo –.

La razón del cambio de nombre es histórica. Cuando Django fue lanzado al público por primera vez, poseía un sistema de formularios complicado y confuso, django.forms. Como hacía muy dificultosa la producción de formularios, fue rescrito y llevo por nombre django.newforms. Sin embargo algunas personas siguieron usando el viejo sistema. Cuando Django 1.0 fue lanzado, el viejo django.forms se fue y django.newforms se convirtió en django.forms otra vez.

Lo primero que necesitas para usar el framework de formularios es definir una clase Form para cada formulario HTML que quieras crear, En este caso únicamente queremos un formulario, así que solo necesitamos tener una clase Form. Esta clase puede localizarse en cualquier lugar – incluso podemos ponerla directamente en el archivo de vistas views.py – pero entre la comunidad la convención es ponerla en un archivo separado llamado **forms.py**. Crea este archivo en el mismo directorio que views.py e introduce lo siguiente:

contactos/forms.py

```
from django import forms

class FormularioContactos(forms.Form):
    asunto = forms.CharField()
    email = forms.EmailField(required=False)
    mensaje = forms.CharField()
```

Esto es bastante intuitivo y muy similar a la sintaxis de modelos de Django. Cada campo en el formulario es representado por un tipo de clase Field – CharField y EmailField son únicamente los tipos de campos usados como atributos de la clase

Form. Cada campo es requerido por defecto, para hacer que el campos email sea opcional, necesitas especificar required=False.

Saltemos al intérprete interactivo de Python y veamos lo que esta clase puede hacer (usando python manage.py shell).

La primera cosa que puede hacer es mostrarse a sí misma como HTML:

```
>>> from contactos.forms import FormularioContactos
>>> f = FormularioContactos()
>>> print (f)
<tr><th><label for="id_asunto">Asunto:</label></th><td><input type="text"
name="asunto" id="id_asunto" /></td></tr>
<tr><th><label for="id_email">Email:</label></th><td><input type="text"
name="email" id="id_email" /></td></tr>
<tr><th><label for="id_mensaje">Mensaje:</label></th><td><input type="text"
name="mensaje" id="id_mensaje" /></td></tr>
```

Para mayor accesibilidad, Django agrega una etiqueta <label> a cada campo, La ideas es hacer el comportamiento predeterminado tan optimo como sea posible.

La salida predeterminada se da en forma de tabla, usando una <table> en formato HTML, pero existen algunas otros tipos de salidas, por ejemplo:

```
>>>#Como lista (print f.as_ul(), si usas Python 2 )
>>> print (f.as_ul())
<li><label for="id_asunto">Subject:</label> <input type="text"
name="asunto" id="id_asunto" /></li>
<li><label for="id_email">Email:</label> <input type="text"
name="email" id="id_email" /></li>
<li><label for="id_mensaje">Message:</label> <input type="text"
name="mensaje" id="id_mensaje" /></li>
>>> #Como Párrafo
>>> print (f.as_p())
<p><label for="id_asunto">Subject:</label> <input type="text"
name="asunto" id="id_asunto" /></p>
<p><label for="id_email">Email:</label> <input type="text"
name="email" id="id_email" /></p>
<p><label for="id_mensaje">Mensaje:</label>
```

Observa que las etiquetas <table>, <ul> y <form> no se han incluido; debes definirlas por tu cuenta en la plantilla. Esto te da control sobre el comportamiento del formulario al ser suministrado. Los elementos label sí se incluyen, y proveen a los formularios de accesibilidad “desde fábrica”.

Estos métodos son solo atajos para los casos comunes, en los que es necesario “mostrar el formulario completo”. Sin embargo también puedes mostrar un campo en particular:

```
>>> print  (f['asunto'])
<input type="text" name="asunto" id="id_asunto" />
>>> print (f['mensaje'])
<input type="text" name="mensaje" id="id_mensaje" />
```

La segunda cuestion sobre los objetos Form es como hacer algo de validación de datos. Para validar datos, crea un nuevo objeto Form y pásale un diccionario de datos que vincule los nombres de los campos con los datos:

```
>>> f = FormularioContactos({'asunto': 'Hola', 'email': 'adrian@example.com',
'mensaje': '¡Buen sitio!'})
```

Una vez que asocias datos, con una instancia de Form, haz creado un formulario "vinculado"

```
>>> f.is_bound
True
```

Una instancia de formulario puede estar en uno de dos estados: bound (vinculado) o unbound (no vinculado). Una instancia bound se construye con un diccionario (o un objeto que funcione como un diccionario) y sabe cómo validar y volver a representar sus datos. Un formulario no vinculado (unbound) no tiene datos asociados y simplemente sabe cómo representarse a sí mismo.

Llama al método is_valid() en un cualquier formulario vinculado para saber si lo datos son validos. En el ejemplo hemos pasado un valor valido a cada campo de la clase Form, así que es completamente valido.

```
>>> f.is_valid()
True
```

Aun si no pasamos el campo email, el formulario sigue siendo válido, porque hemos especificado que el campo sea opcional con required=False

```
>>> f = FormularioContactos({'asunto': 'Hola', 'mensaje': '¡Buen sitio!'})
>>> f.is_valid()
True
```

Pero si dejamos ya sea asunto o mensaje, el Formulario ya no será válido:

```
>>> f = FormularioContactos({'asunto': 'Hola'})
>>> f.is_valid()
False
>>> f = FormularioContactos({'asunto': 'Hola', 'mensaje': ''})
>>> f.is_valid()
False
```

También puedes obtener un mensaje más específico sobre los errores de un campo en particular, de esta forma:

```
>>> f = FormularioContactos({'asunto': 'Hola', 'mensaje': ''})
>>> f['mensaje'].errors
[u'Este campo es obligatorio.']
>>> f['asunto'].errors
[]
>>> f['email'].errors
[]
```

Cada Formulario vinculado que instanciemos contiene un atributo con una variable llamada **errors** en forma de diccionario que asocia los nombres de los campos con la lista de mensajes de errores a mostrar.

```
>>> f = FormularioContactos({'asunto': 'Hola', 'mensaje': ''})
>>> f.errors
{'mensaje': [u'Este campo es obligatorio.']}
```

Finalmente, cuando instanciemos un Formulario que contenga datos que han sido encontrados validos, tendremos disponible un atributo llamado cleaned_data. El cual es un diccionario de datos enviados "Limpiamente". Sin embargo el framework de formularios hace más que validar los datos, también los convierte a tipos de datos Python:

```
>>> f = FormularioContactos({'asunto': 'Hola', 'email':'adrian@example.com',
'mensaje': 'Buen sitio!'})
>>> f.is_valid()
True
>>> f.cleaned_data
{'mensaje': u'Buen sitio!', 'email': u'adrian@example.com', 'asunto': u'Hola'}
```

Nuestro formulario de contactos únicamente trata con cadena, las cuales convierte "limpiamente" en objetos Unicode –pero si utilizáramos IntegerField o DateField, el framework de formularios se aseguraría de usar cleaned_data apropiadamente para tratar con enteros en Python o con objetos datetime.date para los campos dados.

Ligando formularios a vistas

Con un básico conocimiento sobre la forma en que trabajan las clases Form, ahora puedes ver cómo usar esta infraestructura para remplazar algo de código repetitivo en la vista contactos() que creamos anteriormente. Esta es la forma en que podemos rescribir la vista contactos(), para usar el framework de formularios de Django:

contactos/views.py

```
from django.shortcuts import render
from contactos.forms import FormularioContactos

def contactos(request):
    if request.method == 'POST':
        form = FormularioContactos(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['asunto'],
                cd['mensaje'],
                cd.get('email', 'noreply@example.com'),
                    ['siteowner@example.com'],
             )
            return HttpResponseRedirect('/contactos/gracias/')
    else:
        form = FormularioContactos()
    return render(request, 'formulario-contactos.html', {'form': form})
```

contactos/templates/formulario-contactos.html

```
<html>
<head>
    <title>Contactanos</title>
</head>
<body>
     <h1>Contactanos</h1>
```

```
{% if form.errors %}
    <p style="color: red;">Por favor corrige lo siguiente:
    </p>
{% endif %}
    <form action="" method="post">{% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Enviar">
    </form>
</body>
</html>
```

Trata de ejecutar esto localmente. Carga el formulario, envíalo sin datos, ahora introduce una dirección de correo electrónica no valida, y finalmente envíalo con los datos correctamente.

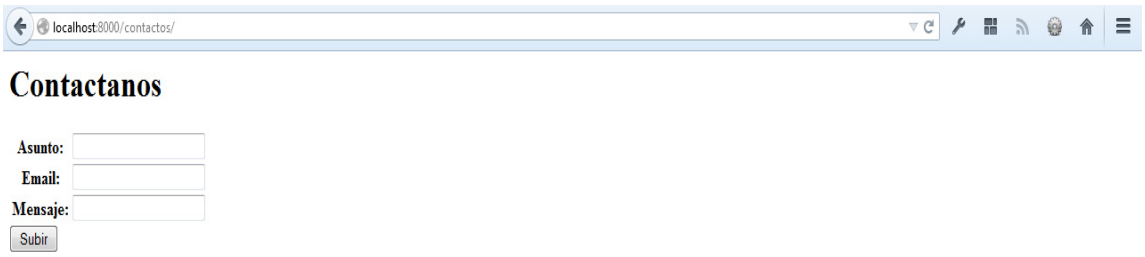


Figura 7-4. Ejemplo de un formulario para contactos usando la clase form.

¡Observa cuanto código repetitivo hemos quitado! El framework de formularios maneja la forma de mostrar el HTML, la validación, la limpieza de datos y se encarga de volver a mostrar el formulario con los errores resaltados.

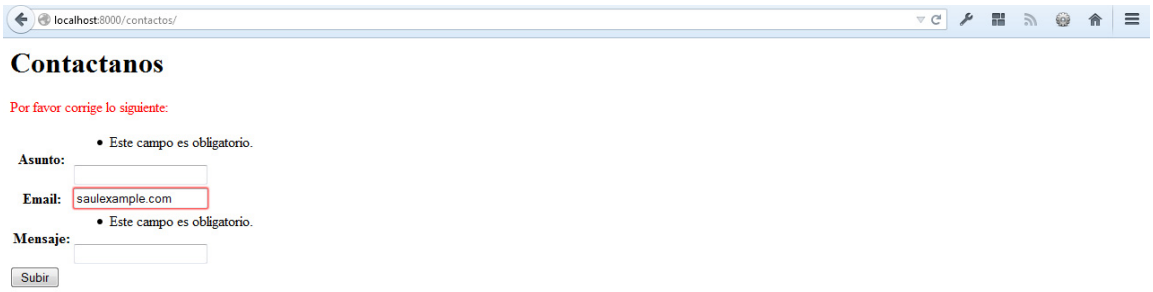


Figura 7-5. Formulario para contactos usando la clase form y la validación automáticas.

Advertencia: Aunque dependiendo de la configuración de tu servidor de email, puedes obtener un mensaje de error cuando llames a send_mail(), pero ese es otro asunto.

Enviar emails usando django

Aunque Python hace relativamente fácil el envió de email, usando el módulo smtplib, Django ofrece un par de contenedores livianos sobre él. Estos contenedores hacen que el envió de emails sea extraordinariamente rápido, especialmente en el desarrollo de tu proyecto, cuando necesitas probar el envió correcto de emails, por lo que Django provee soporte para plataformas que no pueden utilizar SMPT.

El código se localiza en el modulo django.core.mail.

CONFIGURAR UN SERVIDOR DE CORREO EN DJANGO

En Django puedes enviar un email en dos líneas, usando el metodo send_mail():

Primero inicia el intérprete interactivo con el comando python manage.py shell:

```
>>>from django.core.mail import send_mail
>>>send_mail('Este es el argumento', 'Aquí va el mensaje.',
'administrador@example.com',    ['para@example.com'], fail_silently=False)
```

El correo se envía usando el servidor SMPT, con el puerto y el host especificado en el archivo de configuración setting.py, mediante EMAIL_HOST y EMAIL_PORT, mientras que las variables EMAIL_HOST_USER y EMAIL_HOST_PASSWORD se usan para autentificarte con el servidor SMPT si así se requiere, por otra parte EMAIL_USE_TLS y EMAIL_USE_SSL se utilizan para controlar las conexiones seguras y por ultimo EMAIL_BACKEND se utiliza para configurar el servidor de correo a utilizar.

Por omisión Django utiliza SMTP, como la configuración por defecto. Si quieres especificarla explícitamente usa lo siguiente en el archivo de configuraciones:

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

Un servidor de correo usando la terminal

La manera más fácil de configurar el envió de email de forma local es utilizando la consola. De esta forma el servidor de correo dirige todo el email a la salida estándar, permitiéndote revisar el contenido del correo en la terminal.

Solo necesitas especificar el manejador de correo, usando la siguiente configuración:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

De esta forma los emails son redirigidos a la salida estándar de la terminal, para que los puedas visualizar.

Un servidor de correo “bobo”

Así como se pueden recibir correos mediante la terminal, es posible también configurar un servidor de correo “bobo”, que como su nombre sugiere solo simula el envió de correos de verdad.

Para especificarlo, solo usa la siguiente configuración:

```
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'
```

Este servidor de correo, no está diseñado para su uso en producción –ya que solo provee una forma conveniente de enviar emails que puede ser usado en desarrollo de forma local.

Otra forma de utilizar un servidor SMPT "tonto" que reciba los emails localmente y puedas visualizarlos en la terminal, es usando una clase incorporada en Python que inicia un servidor de correo con un solo comando:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Este comando inicia el servidor de correo en el puerto 1025, en el 'host' localhost. Simplemente imprime en la salida estándar las cabeceras y el cuerpo de los emails, así que solo necesitas configurar EMAIL_HOST y EMAIL_PORT respectivamente en el archivo settings.py.

Cambiando la forma en que los campos son renderizados

Probablemente la primer cosa que notaras cuando renderices el formulario localmente sea que el campo mensaje se visualiza como un <input type="text">, y quisieras que fuera un <textarea>. Podemos arreglar esto, configurando el *widget* del campo.

contactos/forms.py

```
from django import forms

class FormularioContactos (forms.Form):
    asunto = forms.CharField()
    email = forms.EmailField(required=False)
    mensaje = forms.CharField(widget=forms.Textarea)
```

El framework de formularios separa la lógica de la presentación, para cada campo en un conjunto de *widgets*. Cada tipo de campo tiene un widget por defecto, pero puedes sobrescribirlo fácilmente, o proporcionar uno nuevo de tu creación.

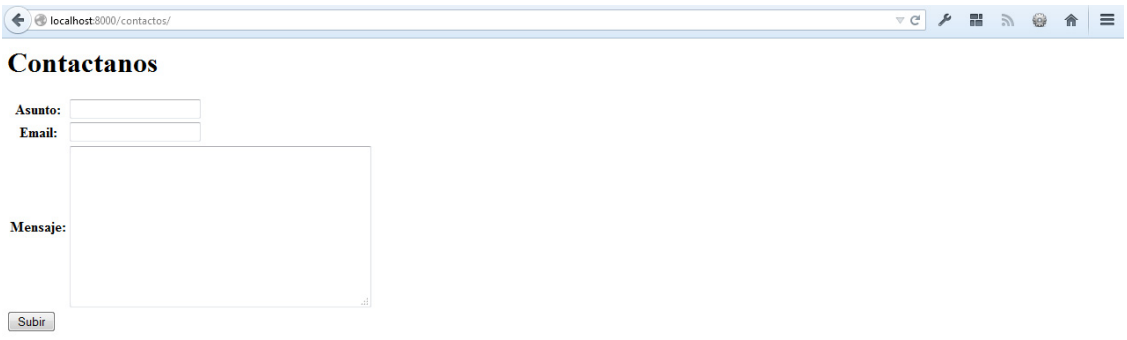


Figura 7-6. Formulario usando un widget textarea.

Piensa en las clases **Field** como las encargadas de la *lógica de validación*, mientras que los **widgets** se encargan de la *lógica de presentación*.

Configurar una longitud máxima

Una de las necesidades más comunes en cuanto a validación es comprobar que un campo tenga un cierto tamaño, es decir que acepte un máximo de caracteres. Como seguimos perfeccionando nuestro Formulario de contactos, sería bueno limitar el asunto a 100 caracteres. Todo lo que tenemos que hacer es agregarle un valor máximo a la opcion max_length de un campo CharField así:

contactos/forms.py

```
from django import forms

class FormularioContactos(forms.Form):
    asunto = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    mensaje = forms.CharField(widget=forms.Textarea)
```

Un argumento opcional min_length (mínimo requerido), también está disponible.

Especificar valores iníciales

Como una mejora mas para nuestro formulario, vamos a agregarle *valores iníciales* al campo asunto algo así como; "¡Adoro este sitio!" (Un poco de sugestión mental, no le hará daño a nadie.) Para hacer esto, usamos un argumento initial en la definición misma de la instancia del formulario;

contactos/views.py

```
def contactos(request):
    if request.method == 'POST':
        form = FormularioContactos(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['asunto'],
                cd['mensaje'],
                cd.get('email', 'noreply@example.com'),
                    ['siteowner@example.com'],
             )
            return HttpResponseRedirect('/contactos/gracias/')
    else:
        form = FormularioContactos(initial={'asunto': '¡Adoro tu sitio!'})
    return render(request, 'formulario-contactos.html', {'form': form})
```

Ahora el formulario inicial mostrara un mensaje en el campo asunto, cada vez que el formulario sea cargado.

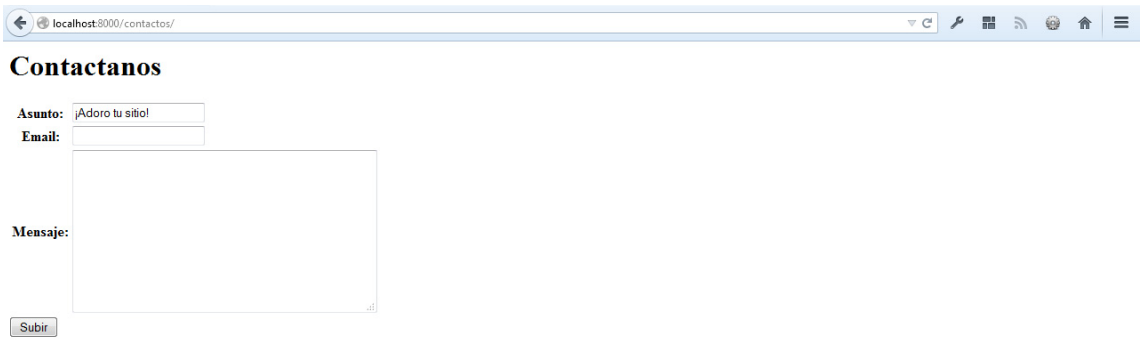


Figura 7-7. Formulario mostrando valores iníciales.

Observa que existe una diferencia entre pasar datos *iníciales* y pasar datos *vinculados* a un formulario. La gran diferencia es que solo estamos pasando datos *iníciales*, entonces el formulario *no está vinculado*, lo cual da a entender que no existen mensajes de error.

Nuestras propias reglas de validación

Imagina que hemos lanzado al público nuestro formulario de comentarios, y los correos electrónicos han empezado a llegar a montones, y nos encontramos con un problema: algunos mensajes vienen con sólo una o dos palabras, es poco probable que tengan algo interesante que decir. Por lo que decidimos adoptar una nueva política de validación: cuatro palabras o más, por favor.

Hay varias formas de insertar nuestras propias reglas de validación en un formulario de Django. Si vamos a usar nuestra regla una y otra vez, podemos crear un nuevo tipo de campo. Sin embargo, la mayoría de las validaciones que agreguemos serán de un solo uso, y pueden agregarse directamente en el propio formulario.

En este caso, necesitamos validación adicional sobre el campo mensaje, así que debemos agregar un método clean_mensaje() a nuestro formulario:

contactos/forms.py

```
from django import forms

class FormularioContactos(forms.Form):
    asunto = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    mensaje = forms.CharField(widget=forms.Textarea)

    def clean_mensaje(self):
        mensaje = self.cleaned_data['mensaje']
        num_palabras = len(mensaje.split())
        if num_palabras < 4:
            raise forms.ValidationError("¡Se requieren mínimo 4 palabras!")
        return mensaje
```

El sistema de formularios de Django, automáticamente busca cualquier método que empiece con clean_ y termine con el nombre del campo. Si cualquiera de estos métodos existe, este será llamado durante la validación.

Específicamente, el método clean_mensaje() es llamado *después* de la validación lógica para el campo por defecto (en este caso, la validación lógica de el campo CharField es obligatoria). Dado que los datos del campo ya han sido parcialmente procesados, necesitamos obtenerlos desde el diccionario self.cleaned_data del formulario. Por lo que no debemos preocuparnos por comprobar que el valor exista y

que no esté vacio; ya que el validador se encargara de realizar este trabajo por defecto.

Usamos una combinación de len() y split() para contar la cantidad de palabras. Si el usuario ha ingresado pocas palabras (menos de 4), lanzamos un error forms.ValidationError. El texto que lleva esta excepción se mostrará al usuario como un elemento más de la lista de errores.
Es importante que retornemos explícitamente el valor limpio del campo al final del método.

Esto nos permite modificar el valor (o convertirlo a otro tipo de Python) dentro de nuestro método de validación. Si nos olvidamos de retornarlo, se retornará None y el valor original ser perderá.

Como especificar Etiquetas

Por defecto las etiquetas HTML autogeneradas por los formularios, son creadas para remplazar los guiones bajos con espacios y con mayúsculas la primera letra de una palabra. – Así por ejemplo la etiqueta para el campo email es "Email" (¿Te suena familiar? es el mismo algoritmo que Django usa en los modelos, para calcular los valores predeterminados de cada campo usando verbose_name, los cuales cubrimos en el *capítulo 5*)

Así como personalizamos los modelos en Django, también podemos personalizar las etiquetas para un campo determinado. Usando para ello la etiqueta label, así:

contactos/forms.py

```
from django import forms

class FormularioContactos(forms.Form):
    asunto = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, label='Tu correo electronico')
    mensaje = forms.CharField(widget=forms.Textarea)

    def clean_mensaje(self):
        mensaje = self.cleaned_data['mensaje']
        num_palabras = len(mensaje.split())
        if num_palabras < 4:
            raise forms.ValidationError("¡Se requieren mínimo 4 palabras!")
        return mensaje
```

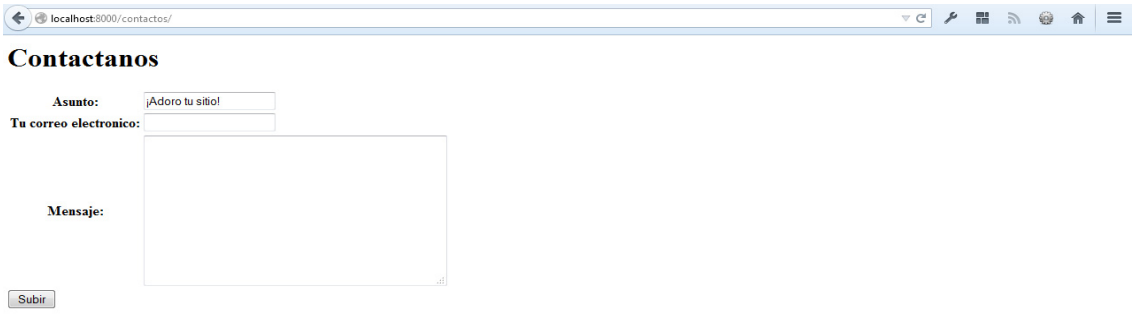


Figura 7-8. Formulario mostrando etiquetas personalizadas.

Diseño de formularios personalizados

La plantilla que usamos formulario-contactos.html usa el formato para tablas {{ form.as_table }} para mostrar el formulario, pero podemos visualizar el formulario de otras maneras, para tener un control más especifico sobre la presentación.

La forma más rápida de personalizar la presentación de un formulario es usando CSS (hojas de estilos). En particular, la lista de errores puede dotarse de mejoras visuales, y el elemento <ul class="errorlist"> tiene asignada una clase para ese propósito. El CSS a continuación hace que nuestros errores salten a la vista, agrega lo siguiente a formulario-contactos.html:

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
}
.errorlist li {
    background-color: red;
    color: white;
    display: block;
    font-size: 10px;
    margin: 0 0 3px;
    padding: 4px 5px;
}
</style>
```

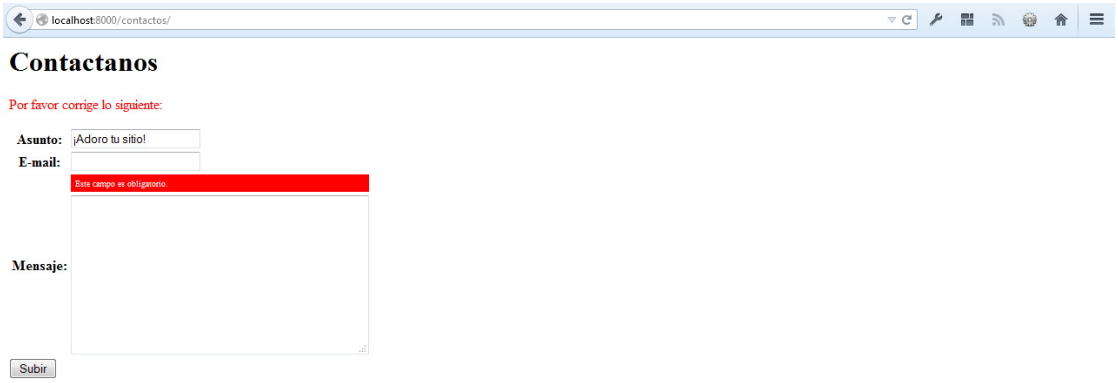


Figura 7-9. Formulario mostrando el resaltado de errores.

Si bien es conveniente que el HTML del formulario sea generado por nosotros, en muchos casos la disposición por defecto queda bien en nuestra aplicación. {{ form.as_table }} y similares son atajos útiles que podemos usar mientras desarrollamos nuestra aplicación, pero todo lo que concierne a la forma en que nuestro formulario es representado puede ser sobrescrito, casi siempre desde la plantilla misma.

Cada widget de un campo (<input type="text">, <select>, <textarea>, o similares) puede generarse individualmente accediendo a {{ form.fieldname }} en la plantilla y cualquier error asociado con un campo está disponible mediante {{ form.fieldname.errors }}. Con esto en mente, podemos construir nuestras propias plantillas personalizada para nuestro formulario de contactos, con el siguiente código:

contactos/templates/formulario-contactos.html

```
<html>
<head>
   <title>Contactanos</title>
</head>

<body>
   <h1>Contactanos</h1>

 {% if form.errors %}
   <p style="color: red;">
   Por favor corrige lo siguiente:
   </p>
{% endif %}

   <form action="" method="post">{% csrf_token %}
      <div class="field">
         {{ form.asunto.errors }}
            <label for="id_asunto">Asunto:</label>
         {{ form.asunto }}
      </div>

      <div class="field">
         {{ form.email.errors }}
            <label for="id_email">E-mail:</label>
        {{ form.email }}
      </div>

      <div class="field">
         {{ form.mensaje.errors }}
            <label for="id_mensaje">Mensaje:</label>
         {{ form.mensaje }}
      </div>

      <input type="submit" value="Enviar">
   </form>
</body>
</html>
```

{{ form.mensaje.errors }} muestra un <ul class="errorlist"> si se presentan errores y muestra una cadena de caracteres en blanco si el campo es válido (o si el formulario no está vinculado). También podemos tratar a la variable form.mensaje.errors como a un booleano o incluso iterar sobre el mismo como una lista, por ejemplo:

```
<div class="field{% if form.mensaje.errors %} errors{% endif %}">
   {% if form.mensaje.errors %}
      <ul>
          {% for error in form.mensaje.errors %}
            <li><strong>{{ error }}</strong></li>
      {% endfor %}
      </ul>
   {% endif %}
   <label for="id_mensaje">Mensaje:</label>
{{ form.mensaje }}
</div>
```

En caso de errores de validación, esto agrega a la clase “errors” el contenedor y muestra los errores en una lista ordenada.

¿Qué sigue?

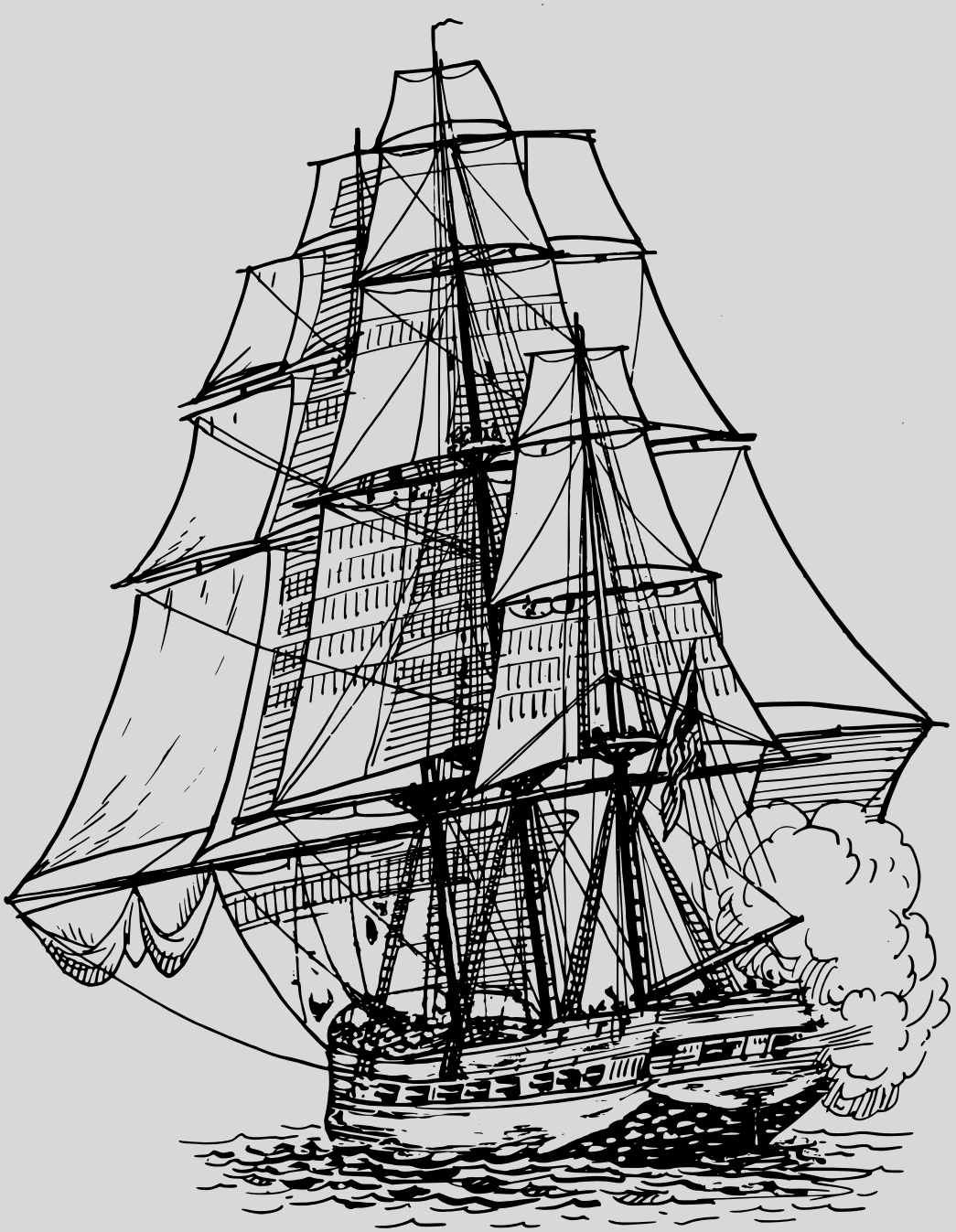
Este capítulo concluye con el material introductorio de este libro. – el también denominado **“nivel básico”**. La siguiente sección del libro, en especial los capítulos 8 al 12 tratan con más detalle sobre el uso avanzado de Django, incluyendo como desplegar una aplicación Django *capítulo 12*.

Luego de estos primeros siete capítulos, deberías saber lo suficiente como para comenzar a escribir tus propios proyectos en Django. El resto del material de este libro te ayudará a completar las piezas faltantes a medida que las vallas necesitando.

Comenzaremos el *capítulo 8* retrocediendo un poco, volviendo atrás para darle una mirada más de cerca a las vistas y a los URLconfs introducidos por primera vez en el *capítulo 3*.

PARTE 2

Nivel avanzado



CAPÍTULO 8

Vistas avanzadas y URLconfs

Un esquema limpio y elegante en una URL, es un detalle importante en una aplicación Web de alta calidad, Django te permite **diseñar las URL** que quieras, sin limitaciones del framework, por lo que no encontraras requerimientos tipo .php o .cgi. El sistema de URLconf que usa Django estimula a generar URLs agradables, haciendo más sencillo el usarlas, que él no usarlas[1].

En él *capítulo 3*, explicamos las bases sobre el funcionamiento de las funciones vista de Django y las URLconfs. En este capítulo conoceremos en detalle algunas funcionalidades más avanzadas en estas dos partes del framework.

Trucos de URLconf

No hay nada de "especial" con las URLconfs – como cualquier otra cosa en Django, son sólo código Python. Puedes aprovecharte de esto de muchas maneras, como se describe más adelante.

Importación de funciones de forma efectiva

Considera esta URLconf, que se basa en el ejemplo del *capítulo 3*:

```
from django.conf.urls import url
from misitio.views import hola, fecha_actual, horas_adelante

urlpatterns = [
    url(r'^hola/$', hola),
    url(r'^fecha/$', fecha_actual),
    url(r'^fecha/mas/(\d{1,2})/$', horas_adelante),
]
```

Como se explicó en el *capítulo 3*, cada entrada en una URLconf debe incluir una función vista asociada, que se pasa directamente como un método. Esto significa que es necesario importar cada una de las funciones de vista o views en la parte superior del módulo.

Sin embargo a medida que las aplicaciones Django crecen en complejidad, sus URLconf crecen también, por lo que mantener esos import puede ser tedioso de manejar. (Por cada nueva función vista, tienes que recordar importarla y la declaración de importaciones tiende a volverse demasiado larga si se utiliza este método). Es posible evitar esa tarea tediosa importando el módulo views directamente.

[1] Para más información sobre URLs elegantes, consulta el artículo titulado Cool URIs don't change (http://www.w3.org/Provider/Style/URI) de el creador de la World Wide Web; Tim Berners-Lee, donde encontraras excelentes argumentos a favor de las URLs limpias y usables.

Este ejemplo de URLconf es equivalente al anterior:

```
from django.conf.urls import url
from misitio import views

urlpatterns = [
    url(r'^hola/$', views.hola),
    url(r'^fecha/$', views.fecha_actual),
    url(r'^fecha/mas/(d{1,2})/$', views.horas_adelante),
]
```

Como puedes observar la sintaxis para la variable urlpatterns, siempre es una lista Python de una instancia de una url().

Django ofrece otra forma de especificar la función vista para un patrón en particular en la URLconf: se le puede pasar un string que contiene el nombre del módulo y el nombre de la función en lugar del método.

Continuando con el ejemplo:

```
from django.conf.urls import url

urlpatterns = [
    url(r'^hola/$', 'misitio.views.hola'),
    url(r'^fecha/$', 'misitio.views.fecha_actual'),
    url(r'^fecha/mas/(d{1,2})/$', 'misitio.views.horas_adelante'),
]
```

(Nota que los nombres de las vistas están entre comillas. Estamos usando 'misitio.views.hola' – con comillas, en lugar de usar misitio.views.hola directamente.)

Al usar esta técnica ya no es necesario importar las funciones vista; Django importa automáticamente la función vista apropiada, la primera vez que sea necesite, según la cadena que describa el nombre y la ruta de la función vista.

Casos especiales de URLs en modo Debug

Otra forma muy común de construir urlpatterns de forma dinámica, es simplemente comprobando el valor de la configuración DEBUG en tiempo de ejecución, para alterar el comportamiento de las URLconf mientras se desarrolla en el modo de depuración de Django así:

```
from django.conf import settings
from django.conf.urls import url

from misitio import views

urlpatterns =[
    url(r'^$', views.indice),
    url(r'^(\d{4})/([a-z]{3})/$', views.archivos_mes),
]

if settings.DEBUG:
urlpatterns += [
    url (r'^debuginfo/$', views.debug),
]
```

En este ejemplo, la URL /debuginfo/ sólo estará disponible si tu configuración DEBUG tiene el valor True.

Trabajar en modo DEBUG, significa sobre todo depurar, por lo que Django, nos provee de herramientas para manejar los casos más comunes del desarrollo Web, como manejar archivos estáticos (hojas de estilo, java script) y archivos media (imágenes, pdf), de forma local. Django hace una gran distinción en el manejo de estos dos tipos de contenido, mientras que se encarga de servir los archivos estáticos de forma automática, para servir los archivos media de forma local, es necesario habilitar una vista opcional y enlazarla a una URLconf en modo DEBUG.

```
from django.views.static import serve
from django.conf import settings
from django.conf.urls import url

from misitio import views

urlpatterns =[
    url(r'^$', views.indice),
    url(r'^(\d{4})/([a-z]{3})/$', views.archivos_mes),
]

if settings.DEBUG:
urlpatterns += [
    url(r'^media/(?P<path>.*)$', serve,
        {'document_root': settings.MEDIA_ROOT,
}),
]
```

En este ejemplo, llamamos a la vista serve, que pertenece al paquete static, que se encarga de servir directorios, solo le pasamos la ruta especificada en MEDIA_ROOT, la cual contiene la raíz de los archivos media, esta vista se encargara de servir archivos (por ejemplo imágenes, videos) siempre y cuando el modo DEBUG este activado. En este ejemplo la variable MEDIA_URL asume que el valor es 'media', aunque podemos cambiarlo segun nuestras necesidades. De esta forma la URL /media/ sólo estará disponible si la configuración DEBUG tiene asignado el valor True.

☠ **Advertencia:** Esta vista es ineficiente y no debe ser usada en producción, por lo que asegúrate de usarla solo en el desarrollo de tus aplicaciones de forma local. Para servir archivos en producción utiliza un servidor dedicado, consulta él *capítulo 12*, para conocer algunos servidores que te pueden ayudar en esa tarea.

Usar grupos con nombre

Hasta ahora en todos nuestros ejemplos URLconf hemos usado, grupos de expresiones regulares *sin nombre* – es decir, ponemos paréntesis en las partes de la URL que queremos capturar y Django le pasa ese texto capturado a la función vista como un argumento posicional. En un uso más avanzado, es posible usar grupos de expresiones regulares *con nombre* para capturar partes
de la URL y pasarlos como argumentos *clave* a una vista.

ARGUMENTOS CLAVES VS. ARGUMENTOS POSICIONALES

A una función de Python se la puede llamar usando argumentos clave o argumentos posicionales – y, en algunos casos, los dos al mismo tiempo. En una llamada por argumentos clave, se especifican los nombres de los argumentos junto con los valores que se le pasan. En una llamada por argumento posicional, sencillamente pasas los argumentos sin especificar explícitamente qué argumento concuerda con cual valor; la asociación está implícita en el orden de los argumentos.

Por ejemplo, considera esta sencilla función:

```
def venta(articulo, precio, cantidad):
    print ("Vendidos: %s unidad(es) de %s a %s" % (cantidad, articulo, precio))
```

Para llamarla con argumentos posicionales, se especifican los argumentos en el orden en que están listados en la definición de la función:

```
venta('Calcetines', '$2.50', 6)
```

Para llamarla con argumentos de palabra clave, se especifican los nombres de los argumentos junto con sus valores. Las siguientes sentencias son equivalentes:

```
venta(articulo='Calcetines', precio='$2.50', cantidad=6)
venta(articulo='Calcetines', cantidad=6, precio='$2.50')
venta(precio='$2.50', articulo='Calcetines', cantidad=6)
venta(precio='$2.50', cantidad=6, articulo='Calcetines')
venta(cantidad=6, articulo='Calcetines', precio='$2.50')
venta(cantidad=6, precio='$2.50', articulo='Calcetines')
```

Finalmente, se pueden mezclar los argumentos posicionales y por palabra clave, siempre y cuando los argumentos posicionales estén listados antes que los argumentos por palabra clave.

Las siguientes sentencias son equivalentes a los ejemplos anteriores:

```
venta('Calcetines', '$2.50', cantidad=6)
venta('Calcetines', precio='$2.50', cantidad=6)
venta('Calcetines', cantidad=6, precio='$2.50')
```

En las expresiones regulares de Python, la sintaxis para los grupos de expresiones regulares con nombre es (?P<nombre>patrón), donde nombre es el nombre del grupo y patrón es algún patrón a buscar.

Aquí hay un ejemplo de URLconf que usa grupos sin nombre, en el primer patrón captura el año, para mostrar una lista de libros por año, en el segundo captura el año y el mes, para mostrar una lista de libros de acuerdo a un año y mes en especifico, el ultimo muestra una lista de libros de acuerdo al año, el mes y el día:

```
from django.conf.urls import url
from libros import views

urlpatterns = [
    url(r'^libros/(\d{4})/$', views.libros_año),
    url(r'^libros/(\d{4})/(\d{2})/$', views.libros_mensuales),
    url(r'^libros/(\d{4})/(\d{2})/(\d{2})/$', views.libros_diarios),
]
```

Aquí está la misma URLconf, reescrita para usar grupos con nombre:

```
from django.conf.urls import url
from libros import views

urlpatterns = [
    url(r'^libros/(?P<año>\d{4})/$', views.libros_año),
    url(r'^libros/(?P<año>\d{4})/(?P<mes>\w{3})/$', views.libros_mes),
    url(r'^libros/(?P<año>\d{4})/(?P<mes>\w{3})/(?P<dia>\d{2})/$', views.libros_dia),
]
```

Esto produce exactamente el mismo resultado que el ejemplo anterior, con una sutil diferencia: se le pasa a las funciones visto los valores capturados como argumentos clave en lugar de argumentos posicionales.

Por ejemplo, con los grupos sin nombre una petición a /libros/2006/03/ resultaría en una llamada de función equivalente a esto:

```
>>>libros_mensuales(request, '2006', '03')
```

Sin embargo, con los grupos con nombre, la misma petición resultaría en esta llamada de función:

```
>>>libros_mensuales(request, año='2006', mes='03')
```

☠ **Advertencia: ¡**Ten cuidado con las "ñ"!, se incluyen solo por legibilidad, no funcionan con Python 2.

En la práctica, usar grupos con nombres hace que tus URLconfs sean un poco más explícitas y menos propensas a errores causados por argumentos – y puedes reordenar los argumentos en las definiciones de tus funciones vista. Siguiendo con el ejemplo anterior, si quisiéramos cambiar las URLs para incluir el mes *antes* del año, y estuviéramos usando grupos sin nombre, tendríamos que acordarnos de cambiar el orden de los argumentos en la vista libros_mes. Si estuviéramos usando grupos con nombre, cambiar el orden de los parámetros capturados en la URL no tendría ningún efecto sobre la vista.

Por supuesto, los beneficios de los grupos con nombre tienen el costo de la falta de brevedad; algunos desarrolladores opinan que la sintaxis de los grupos con nombre es fea y larga. Aún así, otra ventaja de los grupos con nombres es la facilidad de lectura, especialmente para las personas que no están íntimamente relacionadas con las expresiones regulares o con tu aplicación Django en particular. Es más fácil ver lo que está pasando, a primera vista, en una URLconf que usa grupos con nombre.

Una advertencia al usar grupos con nombre en una URLconf es que un simple patrón URLconf no puede contener grupos con nombre y sin nombre. Si haces eso, Django no generará ningún mensaje de error, pero probablemente descubras que tus URLs no se están disparando de la forma esperada.

El algoritmo de combinación/agrupación

Aquí está específicamente el algoritmo que sigue el parser URLconf, con respecto a grupos con nombre vs. grupos sin nombre en una expresión regular:

- Si existe algún argumento con nombre, usará esos, ignorando los argumentos sin nombre.

- Además, pasará todos los argumentos sin nombre como argumentos posicionales.
- En ambos casos, pasará cualquier opción extra como argumentos de palabra clave. Ver la próxima sección para más información.

Pasarle opciones extra a las funciones vista

A veces te encontrarás escribiendo funciones vista que son bastante similares, con tan sólo algunas pequeñas diferencias. Por ejemplo, digamos que tienes dos vistas cuyo contenido es idéntico excepto por la plantilla que utilizan:

urls.py

```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    url(r'^inicio/$', views.vista_inicio),
    url(r'^indice/$', views.vista_indice),
]
```

views.py

```
from django.shortcuts import render
from biblioteca.models import Libro

def vista_inicio(request):
    libros = Libro.objects.all()
        return render(request, 'bienvenidos.html', {'libros': libros})

def vista_indice(request, url):
    libros = Libro.objects.all()
        return render(request, 'indice.html', {'libros': libros})
```

Con este código nos estamos repitiendo y eso no es elegante. Al comienzo, podrías pensar en reducir la redundancia usando la misma vista para ambas URLs, poniendo paréntesis alrededor de la URL para capturarla y comprobar la URL dentro de la vista para determinar la plantilla, como mostramos a continuación:

urls.py

```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    url(r'^inicio/$', views.vista_indice),
    url(r'^indice/$', views.vista_indice),
]
```

views.py

```
from django.shortcuts import render
from biblioteca.models import Libro

def vista_indice(request, url):
    libros = Libro.objects.all()
```

```
    if url == 'inicio':
        plantilla = 'bienvenidos.html'
    elif url == 'indice':
        plantilla = 'indice.html'
    return render(request, plantilla, {'libros': libros})
```

Sin embargo, el problema con esa solución es que acopla fuertemente tus URLs y tu código Si decides renombrar /inicio/ a /bienvenidos/, tienes que recordar cambiar el código de la vista.

La solución elegante involucra un parámetro URLconf opcional. Cada patrón en una URLconf puede incluir un tercer ítem: un diccionario de argumentos de palabra clave para pasarle a la función vista.

Con esto en mente podemos reescribir nuestro ejemplo anterior así:

urls.py

```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    url(r'^inicio/$', views.vista_indice, {'plantilla': 'bienvenidos.html'}),
    url(r'^indice/$', views.vista_indice, {'plantilla': 'indice.html'}),
]
```

views.py

```
from django.shortcuts import render
from biblioteca.models import Libro

def vista_indice(request, plantilla):
    libros = Libro.objects.all()
    return render(request, plantilla, {'libros': libros})
```

Como puedes ver, la URLconf en este ejemplo especifica la plantilla en la URLconf. La función vista la trata como a cualquier otro parámetro.

Esta técnica de la opción extra en la URLconf es una genial forma de enviar información adicional a tus funciones vista sin tanta complicación. Por ese motivo es que es usada por algunas aplicaciones incluidas en Django, más notablemente por el sistema de vistas genéricas, que trataremos en el *capítulo 11*.

La siguiente sección contiene algunas ideas sobre cómo puedes usar la técnica de la opción extra en la URLconf como parte de tus proyectos.

Simulando valores capturados en URLconf

Supongamos que posees un conjunto de vistas que son disparadas vía un patrón y otra URL que no lo es pero cuya lógica de vista es la misma. En este caso puedes "simular" la captura de valores de la URL usando opciones extra de URLconf para manejar esa URL extra con una única vista.

Por ejemplo, podrías tener una aplicación que muestra algunos datos para un día en particular, con URLs tales como:

```
/libros/enero/01/
/libros/enero/02/
/libros/enero/03/
# ...
/libros/abril/30/
/libros/abril/31/
```

A primera vista parece algo complicado, sin embargo esto es simple de manejar – puedes capturar los parámetros en una URLconf como esta (usando sintaxis de grupos con nombre):

```
urlpatterns = [
    url(r'^libros/(?P<mes>\w{3})/(?P<dia>\d{2})/$', views.libros_diarios),
]
```

Y la declaración de la función vista se vería así:

```
def libros_dia(request, mes, dia):
    # ....
```

Este enfoque es simple y directo – no hay nada que no hayamos visto antes. El truco entra en juego cuando quieres agregar otra URL que usa libros_diarios pero cuya URL no incluye un mes ni/o un día.

Por ejemplo, podrías querer agregar otra URL, /libros/favoritos/, que sería el equivalente a /libros/enero/06/. Puedes sacar provecho de las opciones extra de las URLconf de la siguiente forma:

```
urlpatterns = [
    url(r'^libros/favoritos/$', views.libros_dia, {'mes': 'enero', 'dia': '06'}),
    url(r'^libros/(?P<mes>\w{3})/(?P<dia>\d){2}/$', views.libros_dia),
]
```

El detalle genial aquí es que no necesitas cambiar tu función vista para nada. A la función vista sólo le incumbe el obtener los parámetros mes y día – no importa si los mismos provienen de la captura de la URL o de parámetros extra.

Convirtiendo una vista en genérica

Factorizar, es una buena práctica de programación, ya que nos permite aislar las partes comunes del código. Tomemos por ejemplo estas dos funciones Python:

```
def di_hola(nombre_persona):
    print ('Hola, %s' % nombre_persona)

def di_adios(nombre_persona):
    print ('Adios, %s' % nombre_persona)
```

Podemos extraer el saludo para convertirlo en un parámetro así:

```
def saludar(nombre_persona, saludo):
    print ('%s, %s' % (saludo, nombre_persona))
```

Puedes aplicar la misma filosofía a tus vistas Django, usando los parámetros extra de URLconf.

Con esto en mente, puedes comenzar a hacer abstracciones de alto nivel en tus vistas. En lugar de pensar "Esta vista muestra una lista de objetos Libro" y "Esta otra vista muestra una lista de objetos Editor", descubre que ambas son casos específicos de "Una vista que muestra una lista de objetos, donde el tipo de objeto es variable".

Usemos este código como ejemplo:

urls.py

```
from django.conf.urls import  url
from biblioteca import views

 urlpatterns = [
     url(r'^inicio/$', views.lista_libros)
     url(r'^indice/$', views.lista_editores),
 ]
```

views.py

```
from django.shortcuts import render
from biblioteca.models import Libro, Editor

def lista_libros(request):
    lista_libros = Libro.objects.all()
    return render(request, 'biblioteca/lista_libros.html', {'lista_libros': lista_objetos})

def lista_editores(request):
    lista_editores = Editor.objects.all()
    return render(request, 'biblioteca/lista_editores.html', {'lista_editores': lista_objectos})
```

Ambas vistas hacen esencialmente lo mismo: muestran una lista de objetos. Refactoricemos el código para extraer el tipo de objetos en común que muestran:

urls.py

```
from django.conf.urls import  url
from biblioteca import views

urlpatterns = [
    url(r'^lista_libros/$', views.lista_objetos, {'model': models.Libro}),
    url(r'^lista_editores/$', views.lista_objetos, {'model': models.Editor}),
]
```

views.py

```
from django.shortcuts import render

def lista_objectos(request, model):
    lista_objectos = model.objects.all()
    plantilla = 'biblioteca/%s_lista.html' % model.__name__.lower()
    return render(request, plantilla, {'lista_objectos': lista_objectos})
```

Con esos pequeños cambios tenemos de repente, una vista reusable e independiente del modelo. De ahora en adelante, cada vez que necesitemos una lista que muestre un listado de objetos, podemos simplemente rehusar esta vista lista_objectos en lugar de escribir más código.

A continuación, un par de notas acerca de lo que hicimos:

- Estamos pasando las clases de modelos directamente, como el parámetro model. El diccionario de opciones extra de URLconf puede pasar cualquier tipo de objetos Python – no sólo cadenas.

- La línea model.objects.all() es un ejemplo de tipado de pato (*duck typing*): "Si camina como un pato, y habla como un pato, podemos tratarlo como un pato." Nota que el código no conoce de qué tipo de objeto se trata model; el

único requerimiento es que model tenga un atributo objects, el cual a su vez tiene un método all().

- Estamos usando model.__name__.lower() para determinar el nombre de la plantilla. Cada clase Python tiene un atributo __name__ que retorna el nombre de la clase. Esta característica es útil en momentos como este, cuando no conocemos el tipo de clase hasta el momento de la ejecución. Por ejemplo, el __name__ de la clase BlogEntry es la cadena BlogEntry.

- En una sutil diferencia entre este ejemplo y el ejemplo previo, estamos pasando a la plantilla el nombre de variable genérico lista_objectos. Podemos fácilmente cambiar este nombre de variable a lista_libros o lista_editores, pero hemos dejado eso como un ejercicio para el lector.

Debido a que los sitios Web impulsados por bases de datos tienen varios patrones comunes, Django incluye un conjunto de "vistas genéricas" que usan justamente esta técnica para ahorrarte tiempo. Nos ocuparemos de las vistas genéricas incluidas en Django en capítulos siguientes.

Pasando opciones de configuración a una vista

Si estás distribuyendo una aplicación Django, es probable que tus usuarios deseen un cierto grado de configuración. En este caso, es una buena idea agregar puntos de extensión a tus vistas para las opciones de configuración que piensas que la gente pudiera desear cambiar. Puedes usar los parámetros extra de URLconf para este fin.

Una parte de una aplicación que normalmente se hace configurable es el nombre de la plantilla:

```
def una_vista(request, plantilla):
    var = haz_algo()
    return render_to_response(plantilla, {'var': var})
```

Entendiendo la precedencia entre valores capturados vs. opciones extra

Cuando se presenta un conflicto, los parámetros extra de la URLconf tienen precedencia sobre los parámetros capturados. En otras palabras, si tu URLconf captura una variable de grupo con nombre y un parámetro extra de URLconf incluye una variable con el mismo nombre, se usará el parámetro extra de la URLconf.

Por ejemplo, analicemos esta URLconf:

```
from django.conf.urls import  url

urlpatterns = [
    url(r'^libros/(?P<id>\d+)/$', views.lista_libros, {'id': 3}),
]
```

Aquí, tanto la expresión regular como el diccionario extra incluye un id. Tiene precedencia el id fijo especificado en la URL. Esto significa que cualquier petición (por ej. /libros/2/ o /libros/432432/) serán tratados como si id estuviera fijado a 3, independientemente del valor capturado en la URL.

Los lectores atentos notarán que en este caso es una pérdida de tiempo y de tipeo capturar id en la expresión regular, porque su valor será siempre descartado en favor

del valor proveniente del diccionario. Esto es correcto; lo traemos a colación sólo para ayudarte a evitar el cometer este error.

Usando argumentos de vista por omisión

Otro truco común es el de especificar parámetros por omisión para los argumentos de una vista. Esto le indica a la vista qué valor usar para un parámetro por omisión si es que no se especifica ninguno.

Veamos un ejemplo:

urls.py

```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    (r'^libros/$', views.pagina),
    (r'^libros/pagina(?P<num>\d+)/$', views.pagina),
]
```

views.py

```
def pagina(request, num='1'):
# La salida
```

Aquí, ambos patrones de URL apuntan a la misma vista – views.pagina pero el primer patrón no captura nada de la URL. Si el primer patrón es disparado, la función pagina() usará su argumento por omisión para num, "1". Si el segundo patrón es disparado, pagina() usará el valor de num que se haya capturado mediante la expresión regular.

Es común usar esta técnica en combinación con opciones de configuración, como explicamos previamente. Este ejemplo implementa una pequeña mejora al ejemplo de la sección "Pasando opciones de configuración a una vista": provee un valor por omisión para la plantilla:

```
def una_vista(request, plantilla='biblioteca/mi_vista.html'):
    var = haz_algo()
    return render_to_response(plantilla, {'var': var})
```

Manejando vistas en forma especial

En algunas ocasiones tendrás un patrón en tu URLconf que maneja un gran número de URLs, pero necesitarás realizar un manejo especial en una de ellas. En este caso, saca provecho de la forma lineal en la que son procesadas la URLconfs y coloca el caso especial primero.

Por ejemplo, las páginas "agregar un objeto" en el sitio de administración de Django están representadas por la siguiente línea de URLconf:

```
urlpatterns = [
    # ...
    url('^([^/]+)/([^/]+)/add/$', views.add_stage),
    # ...
]
```

Esto se disparará con URLs como /libros/entradas/add/ y /auth/groups/add/. Sin embargo, la página "agregar" de un objeto usuario (/auth/user/add/) es un caso especial – la misma no muestra todos los campos del formulario, muestra dos campos de contraseña, etc. Podríamos resolver este problema tratando esto como un caso especial en la vista, de esta manera:

```
def agregar_estado(request, app_label, model_name):
  if app_label == 'auth' and model_name == 'user':
    # do special-case code
  else:
    # do normal code
```

Pero eso es poco elegante por una razón que hemos mencionado en múltiples oportunidades en este capítulo: Incrusta la lógica de las URLs en la vista. Una manera más elegante sería la de hacer uso del hecho que las URLconfs se procesan desde arriba hacia abajo (en orden descendente):

```
urlpatterns = [
  # ...
  url('^auth/user/add/$', views.user_add_stage),
  url('^([^/]+)/([^/]+)/add/$', views.add_stage),
  # ...
]
```

Con esto, una petición a /auth/user/add/ será manejada por la vista user_add_stage. Aunque dicha URL coincide con el segundo patrón, coincide primero con el patrón ubicado más arriba. (Esto es lógica de corto circuito).

Capturando texto en URLs

Cada argumento capturado es enviado a la vista como una cadena Python, sin importar qué tipo de coincidencia se haya producido con la expresión regular. Por ejemplo en esta línea de URLconf:

```
url(r'^libros/(?P<año>\d{4})/$', views.libros_por_año),
```

El argumento año de views.libros_por_año() será una cadena, no un entero, aun cuando \d{4} sólo coincidirá con cadenas que representen enteros.

Es importante tener esto presente cuando estás escribiendo código de vistas. Muchas funciones incluidas con Python son exigentes (y eso es bueno) acerca de aceptar objetos de cierto tipo. Un error común es intentar crear un objeto datetime.date con valores de cadena en lugar de valores enteros:

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):

TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

Traducido a una URLconf y a una vista, este error se vería así:

```
from django.conf.urls import url
from biblioteca import views
```

```
urlpatterns = [
    url(r'^libros/(\d{4})/(\d{2})/(\d{2})/$', views.libros_dia),
]

import datetime

def librosdiarios(request, año, mes, dia):
    # Lo siguiente lanza un error del "TypeError"
    fecha = datetime.date(añor, mes, dia)
```

En cambio librosdiarios puede ser escrito correctamente de la siguiente forma:

librosdiarios.py

```
def librosdiarios(request, año, mes, dia):
    fecha = datetime.date(int(año), int(mes), int(dia))
```

Observa que int() lanza un ValueError cuando le pasas una cadena que no está compuesta únicamente de dígitos, pero estamos evitando ese error en este caso porque la expresión regular en nuestra URLconf ya se ha asegurado que sólo se pasen a la función vista cadenas que contengan dígitos.

Entendiendo dónde busca una URLconf

Cuando llega una petición, Django intenta comparar los patrones de la URLconf con la URL solicitada como una cadena Python normal (no como una cadena Unicode). Esto no incluye los parámetros de GET o POST o el nombre del dominio. Tampoco incluye la barra inicial porque toda URL tiene una barra inicial.

Por ejemplo, en una petición del tipo http://www.example.com/entrada/ Django tratará de encontrar una coincidencia para entrada/. En una petición para http://www.example.com/entrada/?pagina3 Django tratará de buscar una coincidencia para entrada/.

El método de la petición (por ejemplo POST, GET, HEAD) *no* se tiene en cuenta cuando se recorre la URLconf. En otras palabras, todos los métodos serán encaminados hacia la misma función para la misma URL. Es responsabilidad de una función vista manejar de forma distinta en base al método de la petición.

Abstracciones de alto nivel en las funciones vista

Como se menciona anteriormente, es responsabilidad de una vista manejar de forma distinta cualquier petición, por lo que es necesario tratar de forma distinta los métodos POST, GET.

Veamos cómo construir una vista que trate esto de forma agradable. Considera este diseño:

urls.py

```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    # ...
    url(r'^indice/$', views.indice),
    # ...
]
```

views.py
```
from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render

def indice(request):
    if request.method == 'POST':
        haz_algo_para_post()
        return HttpResponseRedirect('/inicio/')
    elif request.method == 'GET':
        haz_algo_para_get()
        return render(request, 'pagina.html')
    else:
        raise Http404()
```

En este ejemplo, la vista indice() se encarga de manejar tanto peticiones POST como GET, que son totalmente distintas. La única cosa que comparten en común es la misma URL: /inicio/. Como tal es poco elegante manejar ambas peticiones POST y GET en la misma función de vista. Sería más agradable tener dos funciones de vista separadas – una que maneje las peticiones GET y la otra que se encargue de las peticiones POST – por lo que solo debes asegurarte de llamar apropiadamente a la que necesites.

Podemos hacer esto escribiendo una función de vista que delegue la responsabilidad a otra vista, antes o después de ejecutar la lógica definida. Este ejemplo muestra como esta técnica nos puede ayudar a simplificar la vista indice():

views.py
```
from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render

def vista_divida(request, GET=None, POST=None):
    if request.method == 'GET' and GET is not None:
        return GET(request)
    elif request.method == 'POST' and POST is not None:
        return POST(request)
    raise Http404

def peticion_get(request):
    assert request.method == 'GET'
        haz_algo_para_get()
        return render(request, 'pagina.html')

def peticion_post(request):
    assert request.method == 'POST'
        haz_algo_para_post()
    return HttpResponseRedirect('/indice/')
```

urls.py
```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    (r'^indice/$', views.vista_divida, {'GET': views.peticion_get, 'POST':
        views.peticion_post}),
    # ...
]
```

Veamos lo que hicimos:

- Escribimos una nueva vista, llamada vista_divida(), que delega la responsabilidad a dos vistas mas basadas en el tipo de petición mediante el método request.method. Este busca dos argumentos clave, GET y POST, los cuales deben ser *funciones vista*. Si request.method es 'GET', entonces se llama a la vista GET. Si request.method es 'POST', entonces llama a la vista POST. Si request.method es algo como (HEAD, etc.), o si GET o POST no son proporcionados a la función, entonces se lanza un error del tipo Http404 (pagina no encontrada).

- En la URLconf, conectamos /indice/ con vista_divida() y pasamos los argumentos extras – la función de vista para usar GET y POST, respectivamente.

- Finalmente, separamos la vista vista_divida() en dos funciones – peticion_get() y peticion_post(). Esto es mucho más agradable que empaquetar toda la lógica en una simple vista.

Observa que esta función de vista, técnicamente ya no tiene que comprobar request.method, porque la vista_divida() lo hace. (En el momento en que se llame a peticion_post(), por ejemplo, podemos confiar que request.method es 'post'). No obstante, para estar seguros y para que sirva como comprobación, agregamos un assert solo para asegurarnos que request.method haga lo esperado.

Hemos creado una vista genérica agradable que encapsula la lógica y delega el método de petición o request.method a la vista. Nada en este método: vista_divida() ata a nuestra aplicación en particular, por lo que que podemos rehusarla en otros proyectos.

Podemos encontrar una forma de perfeccionar la vista_divida(). Rescribiendo el método, ya que este asume que las vistas GET y POST no toman más argumentos que un request. Entonces ¿Qué pasa si quisiéramos usar vista_divida() con otra vista, por ejemplo para capturar el texto de una URLs, o para que tome argumentos clave opcionales?

Para hacer eso podemos usar una característica genial de Python: que nos permite usar argumentos variables, definidos con asteriscos. Dejaremos primero que el ejemplo lo explique:

```
def vista_divida(request, *args, **kwargs):
    vista_get = kwargs.pop('GET', None)
    vista_post = kwargs.pop('POST', None)
    if request.method == 'GET' and vista_get is not None:
        return vista_get(request, *args, **kwargs)
    elif request.method == 'POST' and vista_post is not None:
        return vista_post(request, *args, **kwargs)
    raise Http404
```

Refactorizemos el método vista_divida para remover los argumentos clave GET y POST, y para poder usar *args y **kwargs (Observa los asteriscos). Esta es una característica de Python que permite a las funciones aceptar de forma dinámica y arbitraria un número de argumentos desconocidos, cuyos nombres no se conocen, hasta en tiempos de ejecución. Con un simple asterisco en la parte superior del parámetro, definimos cualquier argumento *posicional*, por lo que la función se comportara como una tupla. Si usamos dos asteriscos en la parte superior del

parámetro en la definición de la función, cualquier *argumento clave* que pasemos a la función se comportara como un diccionario.

Por ejemplo, con esta función:

```
def vista(*args, **kwargs):
    print ("Los argumentos posicionales son:")
    print (args)
    print ("Los argumentos clave son:")
    print (kwargs)
```

Por convención ***args** se refiere a los parámetros posicionales, mientras que ****kwargs** se refiere a los argumentos clave en una función. Esta es la forma en que trabajaría:

```
>>> vista(1, 2, 3)
Los argumentos posicionales son:
(1, 2, 3)
Los argumentos clave son:
{}
>>> vista(1, 2, name='Adrian', framework='Django')
Los argumentos posicionales son:
(1, 2)
Los argumentos clave son:
{'framework': 'Django', 'name': 'Adrian'}
```

Volviendo a dividir_vista(), puedes usar *args y **kwargs para aceptar *cualquiera* de los argumentos en la función y pasárselos a la vista apropiada. Pero antes de hacer esto, es necesario llamar a kwargs.pop() para obtener los argumentos GET y POST, si están disponibles. (Usamos pop() con un valor predeterminado y *None* para evitar un error del tipo *KeyError* si uno de los otros no está definido.)

Empacando Funciones de Vista

Nuestro truco final toma la ventaja de las técnicas avanzadas de Python. Digamos que encontramos un montón de código repetitivo a lo largo de varias vistas, como en este ejemplo:

```
def vista1(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render(request, 'plantilla1.html')

def vista2(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render(request, 'plantilla2.html')

def vista3(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
        return render(request, 'plantilla3.html')
```

Tenemos aquí, que cada vista empieza comprobando que request.user este autentificado – estos es, que el usuario actual se haya identificado correctamente en el sitio –si no se redirecciona a /accounts/login/. (Observa que aun no cubrimos request.user –El cual veremos en el capítulo14 pero tal como imaginas request.user representa al usuario actual, ya sea anónimo o registrado.)

Seria agradable si quitáramos un poco de código repetitivo de cada una de estas vistas, simplemente marcándolas como vistas que requieren de autentificación. Podemos hacer esto haciendo un wrapper o un contenedor que encapsule estas funcionalidades. Tomate un momento para estudiar lo siguiente:

```
def requiere_login(view):
    def vista_nueva(request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect('/accounts/login/')
        return view(request, *args, **kwargs)
    return vista_nueva
```

La función requiere_login, toma una función vista (view) y retorna una nueva función vista (vista_nueva). La nueva función vista_nueva está definida *dentro* de requiere_login y maneja la lógica comprobando que request.user.is_authenticated() (el usuario este identificado) y delegándolo a la vista original (view).

Ahora, podemos remover la comprobación if not request.user.is_authenticated() de nuestras vistas y simplemente envolviéndolas con requiere_login en nuestra URLconf:

```
from django.conf.urls import urls
from .views import requiere_login, vista1, vista2, vista3

urlpatterns = [
    url(r'^vista1/$', requiere_login(vista1)),
    url(r'^vista2/$', requiere_login(vista2)),
    url(r'^vista3/$', requiere_login(vista3)),
]
```

Esto tiene el mismo efecto que el código anterior, pero con menos código redundante. Acabamos de crear una agradable función genérica – requiere_login() que podemos usar para envolver o contener (wrapping) cualquier vista, para hacer que esta requiera autentificación.

Incluyendo otras URLconfs

Si tu intención es que tu código sea usando en múltiples sitios implementados con Django, debes considerar el organizar tus URLconfs en una manera que permita el uso de inclusiones.

Una URLconf puede, en cualquier punto, "incluir" otros módulos URLconf. Esto se trata, en esencia, de "enraizar" un conjunto de URLs debajo de otras. Por ejemplo, esta URLconf incluye otras URLconfs:

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'^weblog/', include('misitio.blog.urls')),
    url(r'^fotos/', include('misitio.fotos.urls')),
```

```
    url(r'^acerca/$', 'misitio.views.acerca'),
]
```

Existe aquí un detalle importante: en este ejemplo, la expresión regular que apunta a un include() *no* tiene un $ (carácter que coincide con un fin de cadena) pero *si* incluye una barra al final. Cuando Django encuentra include(), elimina todo el fragmento de la URL que ya ha coincidido hasta ese momento y envía la cadena restante a la URLconf incluida para su procesamiento subsecuente.

Continuando con este ejemplo, esta es la URLconf para misitio.blog.urls:

```
from django.conf.urls import url

urlpatterns = [
    url(r'^(\d\d\d\d)/$', 'misitio.blog.views.entrada_año'),
    url(r'^(\d\d\d\d)/(\d\d)/$', 'misitio.blog.views.entrada_mes'),
]
```

Con esas dos URLconfs, veremos aquí cómo serían manejadas algunas peticiones de ejemplo:

Con una peticion a /weblog/2007/: en la primera URLconf, el patrón r'^weblog/' coincide. Debido a que es un include(), Django quita todo el texto coincidente, que en este caso es 'weblog/'. La parte restante de la URL es 2007/, la cual coincide con la primera línea en la URLconf misitio.blog.urls.

Con una peticion a /weblog//2007/: En la primera URLconf, el patrón r'^weblog/' coincide.

Debido a que es un include(), Django quita todo el texto coinciente, que en este caso es weblog/. La parte restante de la URL es /2007/ (con una barra inicial), la cual no coincide con ninguna de las líneas en la URLconf misitio.blog.urls.

/acerca/: Este coincide con el patrón de la vista misitio.views.acerca en la primera URLconf, demostrando que puedes combinar patrones include() con patrones no include().

Otra posibilidad para incluir patrones adicionales en una URL, es usando una lista de instancias de la url(). Por ejemplo, considera esta URLconf.:

```
from django.conf.urls import include, url
from apps.main import views as vista_principal

from credito import views as vista_credito

patrones_extra = [
    url(r'^reportes/(?P<id>[0-9]+)/$', vista_credito.reportes),
    url(r'^cargos/$', vista_credito.cargos),
]

urlpatterns = [
    url(r'^$', vista_principal.indice),
    url(r'^ayuda/', include('apps.ayuda.urls')),
    url(r'^credito/', include(patrones_extra)),
]
```

En este ejemplo la URL /credito/reportes/, será manejada por la vista vista_credito.reportes().

Esto también puede ser usado para remover redundancia en las URLconfs, mediante un simple prefijo en un patrón usado repetidamente. Por ejemplo, considera esta URLconf:

```
from django.conf.urls import url
from  . import views

urlpatterns = [
    url(r'^(?P<pagina_slug>\w+)-(?P<pagina_id>\w+)/historia/$', views.historia),
    url(r'^(?P<pagina_slug>\w+)-(?P<pagina_id>\w+)/editar/$', views.editar),
    url(r'^(?P<pagina_slug>\w+)-(?P<pagina_id>\w+)/discusiones/$',
        views.discusiones),
    url(r'^(?P<pagina_slug>\w+)-(?P<pagina_id>\w+)/permisos/$', views.permisos),
]
```

Podemos perfeccionar esta URLconf declarando un prefijo común una vez, agrupando los sufijos que tienen la misma ruta y excluyendo los que son diferentes:

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^(?P<pagina_slug>\w+)-(?P<pagina_id>\w+)/', include([
        url(r'^historia/$', views.historia),
        url(r'^editar/$', views.editar),
        url(r'^discusiones/$', views.discusiones),
        url(r'^permisos/$', views.permisos),
    ])),
]
```

Cómo trabajan los parámetros capturados con include()

Una URLconf incluida recibe todo parámetro que se haya capturado desde las URLconf padre, por ejemplo:

root urls.py

```
from django.conf.urls import url
urlpatterns = [
    url(r'^(?P<username>\w+)/blog/', include('misitio.urls.blog')),
]
```

misitio/urls/blog.py

```
from django.conf.urls import url
urlpatterns = [
    url(r'^$', 'misitio.views.indice_blog'),
    url(r'^archivos/$', 'misitio.views.archivos_blog'),
]
```

En este ejemplo, la variable capturada username() es pasada a la URLconf incluida y, por lo tanto es pasada a *todas* las funciones vista en dicha URLconf.

Nota que los parámetros capturados son pasados *siempre* a *todas* las líneas en la URLconf incluida, con independencia de si la vista realmente acepta estos parámetros como válidos. Por esta razón esta técnica solamente es útil si estás seguro de que cada vista en la URLconf incluida acepta los parámetros que le estás pasando.

Cómo funcionan las opciones extra de URLconf con include()

De manera similar, puedes pasar opciones extra de URLconf a include() así como puedes pasar opciones extra de URLconf a una vista normal – como un diccionario. Cuando haces esto, *las opciones extra serán pasadas a todas* las líneas en la URLconf incluida.

Por ejemplo, los siguientes dos conjuntos de URLconfs son funcionalmente idénticos.

1. **Conjunto uno:**

urls.py
```
from django.conf.urls import url

urlpatterns = [
    (r'^blog/', include('url-interna'), {'blogid': 3}),
]
```

url-interna.py
```
from django.conf.urls import url

urlpatterns = [
    (r'^archivos/$', 'misitio.views.archivos'),
    (r'^acerca/$', 'misitio.views.acerca'),
    (r'^rss/$', 'misitio.views.rss'),
]
```

2. **Conjunto dos:**

urls.py
```
from django.conf.urls import url

urlpatterns = [
    (r'^blog/', include('url-interna')),
}
```

url-interna.py
```
from django.conf.urls import url

urlpatterns = [
    (r'^archivos/$', 'misitio.views.archivos', {'blogid': 3}),
    (r'^acerca/$', 'misitio.views.acerca', {'blogid': 3}),
    (r'^rss/$', 'misitio.views.rss', {'blogid': 3}),
]
```

Como en el caso de los parámetros capturados (sobre los cuales se explicó en la sección anterior), las opciones extra se pasarán *siempre* a *todas* las URLconf incluidas, sin importar si la vista, realmente acepta estas opciones como válidas. Por esta razón esta técnica es útil sólo si estás seguro que todas las vistas en la URLconf incluida aceptan las opciones extra que les estás pasando.

Resolución inversa de URLs

Una necesidad muy común al trabajar en un proyecto Django es la posibilidad de obtener URLs finales, para incrustar en el contenido generado (vistas y URLs activas, así como URLs para mostrar a los usuarios, etc.) o para manejar el flujo de navegación de el lado del servidor (tal como redirecionamientos, etc.)

Es altamente recomendable evitar codificar en duro las URLs(ya que esta sería una estrategia muy laboriosa, propensa a errores y poco escalable) o tener que idear mecanismos para generar URLs que sean paralelas al diseño descrito por la URLconf, algo semejante podría echar a perder las URLs en algún punto.

En otras palabras, es necesario usar un mecanismo DRY (no te repitas). Entre otras ventajas permitiría la evolución del diseño de URL sin tener que explorar en todas partes del codigo fuente, buscando y remplazando URLs obsoletas.

Como punto de partida para diseñar una URL, podemos empezar usando la información disponible, como puede ser la identificación (el nombre) de la vista a cargo de manejar la URL, otra pieza de información necesaria que podemos anticipar son los tipos (posicional, palabra clave) y los valores y argumentos de la vista, para tomar en cuenta en la URL.

Django ofrece una solución semejante al mapear una URL, únicamente en un solo lugar. Solo la defines en la URLconf y entonces puede usarla en ambas direcciones.

Funciona de dos formas:

1. La primera forma comienza con una petición del usuario/navegador, este llama a la vista correcta de Django y provee cualquier argumento que pueda necesitar así como los valores extraídos del URL.

2. La segunda forma comienza con la identificación de la vista correspondiente de Django más los valores de los argumentos que le son pasados, obtenidos de la URL asociada.

El primero es el usado en las discusiones previas, el segundo es llamado *resolución inversa de URLs*, *búsqueda inversa de URL coincidencias inversas de URLs* o simplemente *URL inversa*.

Django proporciona herramientas para optimizar las coincidencias de URL inversas en las distintas capas donde sean necesarios.

- En las plantillas: Usando la etiqueta de plantillas url
- En el código Python: Usando la función django.core.urlresolvers.reverse
- En código de alto nivel, para relacionar el manejo de URLs de instancias de modelos: por ejemplo el método get_absolute_url en los modelos.

Ejemplos

Considera esta entrada de una URLconf, a la que le hemos agregado un nombre al patron URL, llamado 'libros-anuales', así:

```
from django.conf.urls import url
from biblioteca import views

urlpatterns = [
    #...
    url(r'^libros/([0-9]{4})/$', views.libros_anuales, name='libros-anuales'),
    #...
]
```

De acuerdo al diseño, la URL para la entrada correspondiente al año *nnnn* es /libros/nnnn/.

Para obtener lo mismo en la plantilla usamos este código:

```
<a href="{% url 'libros-anuales' 2014 %}">Libros del 2014</a>
  {# o sin el año en el contexto de la variable de la plantilla: #}
<ul>
  {% for año in lista_anual %}
    <li><a href="{% url 'libros-anuales' año %}">{{ año }} Libros</a></li>
  {% endfor %}
</ul>
```

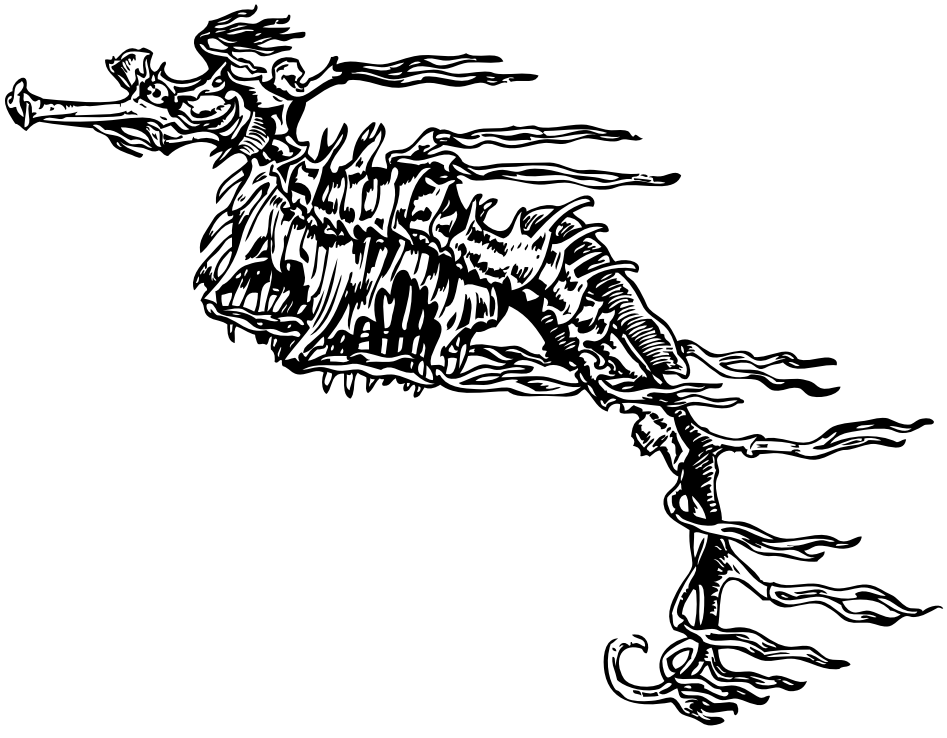
O en el código Python:

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redireccionar_libros_anuales(request):
  # ...
  year = 2014
  # ...
  return HttpResponseRedirect(reverse('libros-anuales', args=(year,)))
```

¿Qué sigue?

Uno de los principales objetivos de Django es reducir la cantidad de código que los desarrolladores deben escribir y en este capítulo hemos sugerido algunas formas en las cuales se puede reducir el código de las vistas y las URLconfs, proporcionándote muchas de las ventajas, tips y trucos para vistas y URLconfs, en el *capítulo 9*, le daremos este tratamiento avanzado al sistema de plantilla de Django.



CAPÍTULO 9

Plantillas avanzadas

Aunque la mayor parte de tu interacción con **el sistema de plantillas** (*templates*) de Django será en el rol de autor, probablemente querrás algunas veces modificar y extender el sistema de plantillas – así sea para agregar funcionalidad, o para hacer tu trabajo más fácil de alguna otra manera.

Este capítulo se adentra en el sistema de plantillas de Django, cubriendo todo lo que necesitas saber, ya sea por si planeas extender el sistema, o por si sólo eres curioso acerca de su funcionamiento. Además cubre la característica de auto-escape, medida de seguridad que seguramente notaras con el paso del tiempo si continuas usando Django.

Si estás tratando de utilizar el sistema de plantillas de Django como parte de otra aplicación, es decir, sin el resto del framework, asegúrate de leer la sección "Configurando el Sistema de plantillas en modo autónomo" más adelante en este mismo capítulo.

Revisión del lenguaje de plantillas

Primero, vamos a recordar algunos términos presentados en el *capítulo 4*:

- **Una plantilla** es un documento de texto, o un string normal de Python marcado con la sintaxis especial del lenguaje de plantillas de Django. Una plantilla puede contener etiquetas de bloque (*block tags*) y variables.

- **Una etiqueta de bloque** es un símbolo dentro de una plantilla que hace algo. Esta definición es así de vaga a propósito. Por ejemplo, una etiqueta de bloque puede producir contenido, servir como estructura de control (una sentencia if o un loop for), obtener contenido de la base de datos, o habilitar acceso a otras etiquetas de plantilla.

  Las etiquetas de bloque deben ser rodeadas por {% y %}

```
{% if is_logged_in %}
    ¡Gracias por identificarte!
{% else %}
    Por favor identificarte.
{% endif %}
```

- **Una variable** es un símbolo dentro de una plantilla que emite un valor.

Las etiquetas de variable deben ser rodeadas por {{ y }}:

```
Mi nombre es {{ nombre }}. Mis apellidos son {{ apellidos }}.
```

- **Un contexto** es un mapeo entre nombres y valores (similar a un diccionario de Python) que es pasado a una plantilla.

Una plantilla *renderiza* un contexto reemplazando los "huecos" que dejan las variables por valores tomados del contexto y ejecutando todas las etiquetes de bloque.

El resto de este capítulo discute las distintas maneras de extender el sistema de plantillas. Aunque primero, debemos dar una mirada a algunos conceptos internos que quedaron fuera del *capítulo 4* por simplicidad.

Procesadores y peticiones de contexto

Cuando una plantilla debe ser renderizada, necesita un contexto. Usualmente este contexto es una instancia de django.template.Context, pero Django también provee una subclase especial: django.template.RequestContext que actúa de una manera levemente diferente. RequestContext agrega muchas variables al contexto de nuestra plantilla – cosas como el objeto HttpRequest o información acerca del usuario que está siendo usado actualmente. El atajo render() crea un RequestContext a menos que explícitamente se le pase una instancia de contexto diferente.

Usa RequestContext cuando no quieras especificar el mismo conjunto de variables una y otra vez en una serie de plantillas. Por ejemplo, considera estas dos vistas:

```
from django.template import loader, Context

def vista_1(request):
    # ...
    t = loader.get_template('plantilla.html')
    c = Context({
        'aplicacion: 'Biblioteca',
        'usuario': request.user,
        'direccion_ip': request.META['REMOTE_ADDR'],
        'mensaje': 'Soy una vista.'
    })
    return t.render(c)

def vista_2(request):
    # ...
    t = loader.get_template('plantilla2.html')
    c = Context({
        'aplicacion: 'Biblioteca',
        'usuario': request.user,
        'direccion_ip': request.META['REMOTE_ADDR'],
        'mensaje': 'Soy otra vista.'
     })
    return t.render(c)
```

A propósito *no* hemos usado el atajo render() en estos ejemplos – manualmente cargamos las plantillas, construimos el contexto y renderizamos las plantillas. Simplemente por claridad, estamos demostrando todos los pasos necesarios.

Cada vista pasa las mismas tres variables – aplicación, usuario y directamente_ip a la plantilla. ¿No sería bueno poder eliminar esa redundancia?

Las ***peticiones de contexto*** (RequestContext) y los ***procesadores de contexto*** (Context Processors) fueron creados para resolver este problema. Los procesadores de contexto te permiten especificar un número de variables que son incluidas automáticamente en cada contexto – sin la necesidad de tener que hacerlo

manualmente en cada llamada a render(). El secreto está en utilizar RequestContext en lugar de Context cuando renderices una plantilla.

La forma de nivel más bajo de usar procesadores de contexto es crear algunos de ellos y pasarlos a RequestContext. A continuación mostramos como el ejemplo anterior puede lograrse utilizando procesadores de contexto:

```
from django.template import loader, RequestContext

    def custom_proc(request):
    "Un procesador de contexto que provee 'aplicacion', 'usuario' y'direcccion_ip'."
    return {
        'aplicacion: 'Biblioteca',
        'usuario': request.user,
        'direccion_ip': request.META['REMOTE_ADDR'],
    }

def vista_1(request):
    # ...
    t = loader.get_template('plantilla1.html')
    c = RequestContext(request, {'mensaje': 'Soy la vista 1.'},
        processors=[custom_proc])
    return t.render(c)

def vista_2(request):
    # ...
    t = loader.get_template('plantilla2.html')
    c = RequestContext(request, {'mensaje': 'Soy la vista 2.'},
        processors=[custom_proc])
    return t.render(c)
```

Inspeccionemos paso a paso este código:

- Primero, definimos una función custom_proc. Este es un procesador de contexto – toma un objeto HttpRequest y devuelve un diccionario con variables a usar en el contexto de la plantilla. Eso es todo lo que hace.

- Hemos cambiado las dos vistas para que usen RequestContext en lugar de Context. Hay dos diferencias en cuanto a cómo el contexto es construido. Uno, RequestContext requiere que el primer argumento sea una instancia de HttpRequest – la cual fue pasada a la vista en primer lugar (request). Dos, RequestContext recibe un parámetro opcional processors, el cual es una lista o una tupla de funciones procesadoras de contexto a utilizar. En este caso, pasamos custom_proc, a nuestro procesador de contexto definido previamente.

- Ya no es necesario en cada vista incluir aplicacion, usuario y dirección_ip cuando construimos el contexto, ya que ahora estas variables son provistas por custom_proc.

- Cada vista *aún* posee la flexibilidad como para introducir una o más variables en el contexto de la plantilla si es necesario. En este ejemplo, la variable de plantilla mensaje es creada de manera diferente en cada una de las vistas.

En él *capítulo 4*, presentamos el atajo render(), el cual nos ahorra tener que llamar a loader.get_template(), luego crear un Context y además, llamar al método render() en la plantilla. Para demostrar el funcionamiento a bajo nivel de los procesadores de contexto, en los ejemplos anteriores no hemos utilizado render(), pero es posible – y preferible utilizar los procesadores de contexto junto a render(). Esto lo logramos mediante el argumento context_instance de la siguiente manera:

```
from django.shortcuts import render
from django.template import RequestContext

def custom_proc(request):
    " Un procesador de contexto que provee 'aplicacion', 'usuario' y 'directamente_ip'."
    return {
        'aplicacion: 'Biblioteca',
        'usuario': request.user,
        'direccion_ip': request.META['REMOTE_ADDR'],
    }

def vista2(request):
    # ...
    return render(request, 'plantilla1.html',
        {'mensaje': 'Soy la vista 1.'},
            context_instance=RequestContext(request, processors=[custom_proc]))

def vista2(request):
    # ...
    return render(request, 'template2.html',
        {'mensaje': 'Soy la vista 2.'},
            context_instance=RequestContext(request, processors=[custom_proc]))
```

Aquí, hemos logrado reducir el código para renderizar las plantillas en cada vista a una sola línea.

Esto es una mejora, pero, evaluando la concisión de este código, debemos admitir que hemos logrado reducir la redundancia en los datos (nuestras variables de plantilla), pero aun así, estamos especificando una y otra vez nuestro contexto. Es decir, hasta ahora el uso de procesadores de contexto no nos ahorra mucho código, si tenemos que escribir procesadores constantemente.

Por esta razón, Django admite el uso de procesadores de contexto *globales*. El parámetro de configuración TEMPLATE_CONTEXT_PROCESSORS designa cuales serán los procesadores de contexto que deberán ser aplicados *siempre* a RequestContext. Esto elimina la necesidad de especificar processors cada vez que utilizamos RequestContext.

TEMPLATE_CONTEXT_PROCESSORS tiene, por omisión, el siguiente valor:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
)
```

Este parámetro de configuración es una tupla de funciones que utilizan la misma interfaz que nuestra función custom_proc utilizada previamente – funciones que toman un objeto HttpRequest como primer argumento, y devuelven un diccionario

de items que serán incluidos en el contexto de la plantilla. Ten en cuenta que los valores en TEMPLATE_CONTEXT_PROCESSORS son especificados como *cadenas*, lo cual significa que estos procesadores deberán estar en algún lugar dentro de tu PYTHONPATH (para poder referirse a ellos desde el archivo de configuración)

Estos procesadores de contexto son aplicados en orden, es decir, si uno de estos procesadores añade una variable al contexto y un segundo procesador añade otra variable con el mismo nombre, entonces la segunda sobre-escribirá a la primera.

Django provee un número de procesadores de contexto simples, entre ellos los que están activos por defecto.

django.core.context_processors.auth

Si TEMPLATE_CONTEXT_PROCESSORS contiene este procesador, cada RequestContext contendrá las siguientes variables:

- ***user***: Una instancia de django.contrib.auth.models.User representando al usuario actualmente autenticado (o una instancia de AnonymousUser si el cliente no se ha autenticado aún).

- ***messages***: Una lista de mensajes (como *string*) para el usuario actualmente autenticado. Detrás del telón, esta variable llama a request.user.get_and_delete_messages() para cada *request*. Este método colecta los mensajes del usuario, y luego los borra de la base de datos.

- ***perms:*** Instancia de django.core.context_processors.PermWrapper, la cual representa los permisos que posee el usuario actualmente autenticado.

En él *capítulo 14* encontrarás más información acerca de usuarios, permisos y mensajes.

django.core.context_processors.debug

Este procesador agrega información de depuración a la capa de plantillas. Si TEMPLATE_CONTEXT_PROCESSORS contiene este procesador, cada RequestContext contendrá las siguientes variables:

- ***debug:*** El valor del parámetro de configuración DEBUG (True o False). Esta variable puede usarse en las plantillas para saber si estás en modo de depuración o no.

- ***sql_queries***: Una lista de diccionarios {'sql': ..., 'time': ...} representando todas las consultas SQL que se generaron durante la petición (*request*) y cuánto duraron. La lista está ordenada respecto a cuándo fue ejecutada cada consulta.

Como la información de depuración es sensible, este procesador de contexto sólo agregará las variables al contexto si las dos siguientes condiciones son verdaderas.

- El parámetro de configuración DEBUG es True

- La solicitud (*request*) viene de una dirección IP listada en el parámetro de configuración

■ **INTERNAL_IPS:** Los lectores astutos se darán cuenta, que si la variable de plantilla debug tiene el valor False, las demás variables de plantillas que dependen de debug, no podrán cargarse en primer lugar.

django.core.context_processors.i18n

Si este procesador está habilitado, cada RequestContext contendrá las siguientes variables:

- *LANGUAGES*: El valor del parámetro de configuración LANGUAGES.
- *LANGUAGE_CODE*: request.LANGUAGE_CODE si existe; de lo contrario, el valor del parámetro de configuración LANGUAGE_CODE.

En el Apéndice D se especifica más información sobre estos parámetros.

django.core.context_processors.request

Si este procesador está habilitado, cada RequestContext contendrá una variable request, la cual es el actual objeto HttpRequest. Observa que este procesador no está habilitado por defecto; tú tienes que activarlo.

Tal vez quieras usarlo, si necesitas que tus plantillas tengan acceso a los atributos de la actual HttpRequest tal como la dirección IP:

```
{{ request.REMOTE_ADDR }}
```

Consideraciones para escribir tus propios procesadores de contexto

Algunos puntos a tener en cuenta:

- Cada procesador de contexto debe ser responsable por la mínima cantidad de funcionalidad posible. Usar muchos procesadores es algo sencillo, es por eso que dividir la funcionalidad de tu procesador de manera lógica puede ser útil para poder reutilizarlos en el futuro.
- Ten presente que cualquier procesador de contexto en TEMPLATE_CONTEXT_PROCESSORS estará disponible en *cada* plantilla cuya configuración esté dictada por ese archivo de configuración, así que trata de seleccionar nombres de variables con pocas probabilidades de entrar en conflicto con nombre de variables que tus plantillas pudieran usar en forma independiente. Como los nombres de variables son sensibles a mayúsculas/minúsculas no es una mala idea usar mayúsculas para las variables provistas por un procesador.
- No importa dónde residan en el sistema de archivos, mientras se hallen en tu ruta de Python de manera que puedas incluirlos en tu variable de configuración TEMPLATE_CONTEXT_PROCESSORS. Habiendo dicho eso, diremos también que la convención es grabarlos en un archivo llamado context_processors.py ubicado en tu aplicación o en tu proyecto.

Escape automático de HTML

Cuando generamos HTML por medio de plantillas, siempre existe el riesgo de incluir variables que contengan caracteres que afecten la salida del HTML. Por ejemplo, considera el siguiente fragmento de una plantilla:

```
Hola, {{ nombre }}.
```

Esto parece inofensivo al principio, ya que solo muestra el nombre de un usuario, pero considera lo siguiente: que pasaría si el usuario introduce su nombre de la siguiente manera:

```
<script>alert('hola')</script>
```

Con este valor, la plantilla renderizaria el nombre así:

```
Hola, <script>alert('hola')</script>
```

Lo cual daría como resultado que el navegador mostrara una caja de alerta. De igual forma, si el nombre contiene símbolos como este '<':

```
<b>usernombre
```

Esto daría como resultado una plantilla renderizada así:

```
Hola, <b>usernombre
```

Lo cual a su vez daría como resultado que el nombre apareciera en negritas.

Claramente, no deberías confiar en todo lo que los usuarios suban ciegamente, ni tampoco deberías permitir que inserten datos directamente en las páginas web, porque algún usuario malicioso podría aprovecharse de huecos como estos, para hacer cosas potencialmente dañinas.

Este tipo de de riesgos de seguridad es llamado ataque Cross Site Scripting (XSS) (Scripting inter-sitio). Consulta el *capítulo 20*, para conocer más sobre seguridad.

Para evitar este problema, tienes dos opciones:

1. Uno, asegúrate de que cada una de las variables “no confiables”, sean pasadas a través de un filtro escape, el cual convierte caracteres potencialmente dañinos en no dañinos. Esta fue la solución por defecto en Django durante los primeros años, pero el problema es que pone toda la responsabilidad en *ti* que como desarrollador/autor de plantillas, debes asegurarte de escapar todo. Es fácil olvidar escapar datos.

2. Dos, puedes tomar ventaja de que Django automáticamente escapa el HTML. El resto de esta sección describe como trabaja el auto-escape.

Por defecto en Django, cada plantilla se encarga automáticamente de escapar la salida de cada etiqueta de variable. Es particular estos cinco caracteres son escapados:

- **<** es convertido a <
- **>**es convertido a >
- **'** (comillas simples) son convertidas a '
- **"** (comillas dobles) son convertidas a "

- **&** es convertido a &

De nuevo, hacemos énfasis en que este comportamiento se da por defecto. Si estas usando el sistema de plantillas, estas protegido.

Como desactivar el escape automático

Si no quieres que los datos sean escapados automáticamente en el sitio, en algún nivel de las plantillas o a nivel de variables, puedes desactivar este comportamiento de diferentes formas.

¿Porque querrías desactivarlo? Porque algunas veces, las variables de plantillas contienen datos que *intentas* renderizar como HTML en crudo, en cuyo caso necesitas que el contenido no sea escapado. Por ejemplo, si quisieras almacenar algunos datos confiables en HTML en la base de datos y quisieras incrustarlos directamente en la plantilla o si quieres usar el sistema de plantillas para producir texto que *no* sea HTML –Tal como un mensaje de email, para una instancia.

Para Variables individuales

Para desactivar el auto-escape para una variable individual, usa el filtro safe:

- Esto será escapado: {{ datos }}
- Esto no será escapado {{ datos|safe }}

Piensa en *safe* como el nombre corto para *safe from further escaping* o *puede ser interpretado de forma segura como HTML* En este ejemplo, si datos contiene '<b>', la salida será:

- Esto sera escapado: <b>
- Esto no sera escapado: <b>

Para bloques de plantillas

Para controlar el auto-escapado de una plantilla, envuelve la plantilla (o solo una sección en particular) con la etiqueta autoescape, de esta forma:

```
{% autoescape off %}
    Hola {{ nombre }}
{% endautoescape %}
```

La etiqueta autoescape toma cualquiera de estos argumentos: on u off. Ocasionalmente, tal vez quieras forzar el auto-escape, cuando este desactivado. Esta es una plantilla de ejemplo:

```
Auto-escape activado por defecto.

Hola {{ nombre }}

{% autoescape off %}
    Esto no será auto-escapado: {{ datos }}.

    Esto tampoco: {{ otros_datos }}
    {% autoescape on %}
        Auto-escape aplicado otra vez: {{ nombre }}
```

```
    {% endautoescape %}

{% endautoescape %}
```

La etiqueta de auto-escape pasa sus efectos a las plantillas que extiende, así como a las plantillas incluidas vía la etiqueta include, tal como lo hacen todos los bloques de etiquetas.

Por ejemplo:

base.html

```
{% autoescape off %}
    <h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
```

child.html

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

{% block title %}Bienvenidos{% endblock %}

{% block content %}{{ saludo }}{% endblock %}
```

Una vez que el auto-escape es desactivado en la plantilla base, también puede ser desactivado en las plantillas hijas, resultando en el siguiente HTML renderizado cuando la variable saludo contiene la cadena Hola!:

```
<h1>Bienvenidos</h1>
<b>¡Hola!</b>
```

NOTA: En general, los autores de plantillas no necesitan preocuparse por auto-escapar mucho. Los desarrolladores del lado de Python (personas que escriben vista y filtros a la medida) necesitan pensar en los casos en los cuales sus datos no sean escapados y marcarlos apropiadamente, pensando en la forma en que trabajan las plantillas.

Si estas creando plantillas, para situaciones en las que no estás seguro si el auto-escape esta activado, puedes agregar un filtro escape a cualquier variable que necesite ser escapada. Cuando el auto-escape esta activado, no hay peligro de el filtro escape o del *doble-escapado* de datos – el filtro escape no afecta las variables que ya han ha sido escapadas.

Escape automático de cadenas literales en argumentos de filtros

Como mencionamos anteriormente, los argumentos de algunos filtros pueden ser cadenas:

```
{{ datos|default:"Esta es una cadena literal." }}
```

Todas las cadenas literales son insertadas *sin* escape automático en la plantilla – actúan como si fueran pasadas por el filtro safe. La razón detrás de todo esto, es que los autores de plantillas tienen el control de lo que pasan dentro de las cadenas

literales, así que se aseguran que el texto sea escapado correctamente cuando la plantilla es escrita.

Por lo que escribirías

```
{{ datos default:"3 &lt; 2" }}
```

En lugar de:

```
{{ datos default:"3 < 2" }}  <-- ¡Mal! No hagas esto.
```

Esto no afecta el comportamiento de los datos, que provienen de la misma variable. El contenido de las variable todavía es escapado automáticamente si es necesario, porque va mas allá de el control del autor de la plantilla.

Detalles internos de la carga de plantillas

En general las plantillas se almacenan en archivos en el sistema de archivos, pero puedes usar cargadores de plantillas personalizados (*template loaders*) para cargar plantillas desde otros orígenes.

Django tiene dos maneras de cargar plantillas:

- ***django.template.loader.get_template(template):*** get_template retorna la plantilla compilada (un objeto Template) para la plantilla con el nombre provisto. Si la plantilla no existe, se generará una excepción TemplateDoesNotExist.

- ***django.template.loader.select_template(template_nombre_list):*** select_template es similar a get-template, excepto que recibe una lista de nombres de plantillas. Retorna la primera plantilla de dicha lista que existe. Si ninguna de las plantillas existe se lanzará una excepción TemplateDoesNotExist.

Como se vio en el *capítulo 4*, cada una de estas funciones usan por omisión el valor de la variable de configuración TEMPLATE_DIRS para cargar las plantillas. Sin embargo, internamente las mismas delegan la tarea pesada a un cargador de plantillas.

Algunos de los cargadores están, de forma predeterminada desactivados, pero puedes activarlos editando la variable de configuración TEMPLATE_LOADERS.

TEMPLATE_LOADERS debe ser una tupla de cadenas, donde cada cadena representa un cargador de plantillas.

Estos son los cargadores de plantillas incluidos con Django:

- ***django.template.loaders.filesystem.load_template_source:*** Este cargador carga plantillas desde el sistema de archivos, de acuerdo a TEMPLATE_DIRS. Por omisión está activo.

- ***django.template.loaders.app_directories.load_template_source:*** Este cargador carga plantillas desde aplicaciones Django en el sistema de archivos. Para cada aplicación en INSTALLED_APPS, el cargador busca un sub-directorio templates. Si el directorio existe, Django buscará una plantilla en el mismo.

- ***django.template.loaders.eggs.load_template_source:*** Este cargador es básicamente idéntico a app_directories, excepto que carga las plantillas desde eggs Python en lugar de hacerlo desde el sistema de archivos. Por omisión este

cargador está desactivado; necesitarás activarlo si estás usando eggs para distribuir tu aplicación.

Esto significa que puedes almacenar plantillas en tus aplicaciones individualmente, facilitando la distribución de aplicaciones Django con plantillas predeterminadas. Por ejemplo si INSTALLED_APPS contiene ('misitio.blog', 'misitio.musica') entonces get_template('foo.html') buscará plantillas en el siguiente orden:

1. /ruta/a/misitio/blog/templates/foo.html
2. /ruta/a/misitio/musica/templates/foo.html

También debes saber, que el cargador realiza una optimización cuando es importado por primera vez: hace cacheo de una lista de paquetes registrados en INSTALLED_APPS , buscando en el sub-directorio templates. Por omisión este cargador está activado.

Django usa los cargadores de plantillas en el orden en el que aparecen en la variable de configuración TEMPLATE_DIRS. Usará cada uno de los cargadores hasta que uno de los mismos tenga éxito en la búsqueda de la plantilla.

Extender el sistema de plantillas

Ahora que entiendes un poco más acerca del funcionamiento interno del sistema de plantillas, echemos una mirada a cómo extender el sistema con código propio.

La mayor parte de la personalización de plantillas se da en forma de etiquetas y/o filtros. Aunque el lenguaje de plantillas de Django incluye muchos, probablemente diseñaras tus propias bibliotecas de etiquetas y filtros que se adapten a tus propias necesidades. Afortunadamente, es muy fácil definir tu propia funcionalidad.

Crear una biblioteca para etiquetas

Ya sea que estés escribiendo etiquetas o filtros personalizados, la primera tarea a realizar es crear una **biblioteca para etiquetas** – un pequeño fragmento de infraestructura con el cual Django puede interactuar (un directorio).

La creación de una biblioteca para etiquetas es un proceso de dos pasos:

- Primero, decidir qué aplicación Django alojará el directorio. Si has creado una aplicación vía manage.py startapp puedes colocarla allí, o puedes crear otra aplicación con el solo fin de alojar la biblioteca.
  Sin importar cuál de las dos rutas tomes, asegúrate de agregar la aplicación a tu variable de configuración INSTALLED_APPS. Explicaremos esto un poco más adelante.

- Segundo, crea un directorio *templatestags* en el paquete de aplicación Django apropiado. Debe encontrarse en el mismo nivel que models.py, views.py, etc. Por ejemplo mira el siguiente directorio:

```
biblioteca/
    __init__.py
    admin.py
    forms.py
    models.py
    templates/
        indice.html
    templatestags/
```

```
        __init__.py
        etiquetas.py
    views.py
```

Crea dos archivos vacíos en el directorio templatetags: un archivo __init__.py (para indicarle a Python que se trata de un paquete que contiene código Python) y un archivo que contendrá tus definiciones personalizadas de etiquetas/filtros. El nombre del segundo archivo es el que usarás para cargar las etiquetas más tarde. Por ejemplo, si tus etiquetas/filtros personalizadas están en un archivo llamado etiquetas.py, entonces deberás escribir lo siguiente en una plantilla:
{% load etiquetas %}

La etiqueta {% load %} examina tu variable de configuración INSTALLED_APPS y sólo permite la carga de bibliotecas para plantillas desde aplicaciones Django que estén instaladas. Se trata de una característica de seguridad; te permite tener en cierto equipo el código Python de varias bibliotecas para plantillas sin tener que activar el acceso a todas ellas para cada instalación de Django.

Si escribes una biblioteca para etiquetas que no se encuentra atada a ningún modelo/vista particular es válido y normal el tener un paquete de aplicación Django que sólo contiene un paquete templatetags. No existen límites en lo referente a cuántos módulos puedes poner en el paquete templatetags. Sólo ten presente que una sentencia {% load %} cargará etiquetas/filtros para el nombre del módulo Python provisto, no el nombre de la aplicación.

Una vez que has creado ese módulo Python, sólo tendrás que escribir un poquito de código Python, dependiendo de si estás escribiendo filtros o etiquetas.

Para ser una biblioteca de etiquetas válida, el módulo debe contener una variable a nivel del módulo llamada register, que sea una instancia de template.Library. Esta instancia de template.Library es la estructura de datos en la cual son registrados todas las etiquetas y filtros.

Así que inserta en la zona superior de tu módulo, lo siguiente:

```
from django import template

register = template.Library()
```

■**Nota:** Para ver un buen número de ejemplos, examina el código fuente de los filtros y etiquetas incluidos con Django. Puedes encontrarlos en django/template/defaultfilters.py y django/template/defaulttags.py, respectivamente. Algunas aplicaciones en django.contrib también contienen bibliotecas para plantillas.

Una vez que hayas creado esta variable register, puedes usarla para crear filtros y etiquetas para plantillas.

Escribir filtros de plantilla personalizados

Los filtros personalizados son sólo funciones Python que reciben uno o dos argumentos:

- El valor de la variable (entrada)

- El valor del argumento, el cual puede tener un valor por omisión o puede ser obviado.

Por ejemplo, en el filtro {{ var|foo:"bar" }} el filtro foo recibiría el contenido de la variable var y el argumento "bar".

Las funciones filtro deben siempre retornar algo. No deben arrojar excepciones, y deben fallar silenciosamente. Si existe un error, las mismas deben retornar la entrada original o una cadena vacía, dependiendo de qué sea más apropiado.

Este es un ejemplo de la definición de un filtro:

```
def cortar(value, arg):
    "Remueva todos los valores que concuerdan con los
    Argumentos de la cadena   dada"

   return value.replace(arg, '')
```

Y este es un ejemplo de su forma de uso:

```
{{ variable|cortar:"0" }}
```

La mayoría de los filtros no reciben argumentos. En este caso, basta con que no incluyas el argumento en tu función:

```
# Únicamente un argumento.
def minusculas(valor):
   "Convierte una cadena a minúsculas"

   return valor.lower()
```

Una vez que has escrito tu definición de filtro, necesitas registrarlo en una instancia de Library, para que esté disponible para el lenguaje de plantillas de Django:

```
register.filter('cortar', cortar)
register.filter('minusculas', minusculas)
```

El método Library.filter() tiene dos argumentos:

1. El nombre del filtro (una cadena)

2. La función filtro propiamente dicha

Si estás usando Python 2.7 o una versión superior, puedes usar register.filter() como un decorador:

```
@register.filter(name='cortar')
def cortar(valor, arg):
   return valor.replace(arg, '')

@register.filter
def minusculas(valor):
   return valor.lower()
```

Si no provees el argumento *name*, como en el segundo ejemplo, Django usará el nombre de la función como nombre del filtro.

Veamos entonces el ejemplo completo de una biblioteca para plantillas, que provee el filtro cortar:

```
from django import template

register = template.Library()

@register.filter(name='cortar')
def cortar(valor, arg):
   return valor.replace(arg, '')
```

Escribir etiquetas de plantilla personalizadas

Las etiquetas son más complejas que los filtros porque las etiquetas pueden implementar prácticamente cualquier funcionalidad.

El *capítulo 4* describe cómo el sistema de plantillas funciona en un proceso de dos etapas: *compilación* y *renderizado*. Para definir una etiqueta de plantilla personalizada, necesitas indicarle a Django cómo manejar ambas etapas cuando llega a tu etiqueta.

Cuando Django compila una plantilla, divide el texto crudo de la plantilla en *nodos*. Cada nodo es una instancia de django.template.Node y tiene un método render(). Por lo tanto, una plantilla compilada es simplemente una lista de objetos Node. Por ejemplo considera esta plantilla:

```
Hola, {{ persona.nombre }}.

{% ifequal nombre.nacimiento hoy %}
   ¡Feliz cumpleaños!
{% else %}
Vaya de seguro recibirás un mensaje de cumpleaños
sorpresa esplendido el día de tu cumpleaños.
{% endifequal %}
```

En un formulario compilado, esta plantilla es representada como una lista de nodos:

- Nodo texto: "Hola, "
- Nodo variable: person.nombre
- Nodo texto: ".\n\n"
- Nodo IfEqual : nombre.nacimiento y hoy

Cuando llamas a render() en una plantilla compilada, la plantilla llama a render() en cada Node() de su lista de nodos, con el contexto proporcionado. Los resultados son todos concatenados juntos para formar la salida de la plantilla. A sí que para definir una etiqueta de plantilla personalizada debes especificar cómo se debe convertir la etiqueta en crudo en un Node (la función de compilación) y definir lo qué hace el método render() del nodo.

En las siguientes secciones explicaremos los pasos necesarios para escribir una etiqueta propia.

Escribir la función de compilación

Para cada etiqueta de plantilla que encuentra, el intérprete (*parser*) de plantillas llama a una función de Python pasándole el contenido de la etiqueta y el objeto parser en sí mismo. Esta función tiene la responsabilidad de retornar una instancia de Node basada en el contenido de la etiqueta.

Por ejemplo, escribamos una etiqueta {% current_time %} que visualice la fecha/hora actuales con un formato determinado por un parámetro pasado a la etiqueta, usando la sintaxis de strftime (consulta http://www.djangoproject.com/r/python/strftime/, para mas detalles). Es una buena idea definir la sintaxis de la etiqueta previamente. En nuestro caso, supongamos que la etiqueta deberá ser usada de la siguiente manera:

```
<p>La fecha actual es {% fecha_actual "%Y-%m-%d %I:%M %p" %}.</p>
```

■**Nota:** Si, esta etiqueta de plantilla es redundante – La etiqueta {% now %} incluida en Django por defecto hace exactamente lo mismo con una sintaxis más simple. Sólo mostramos esta etiqueta a modo de ejemplo.

Para evaluar esta función, se deberá obtener el parámetro y crear el objeto Node:

```
from django import template

register = template.Library()

def fecha_actual(parser, token):
    try:
        # El metodo split_contents() sabe como dividir cadenas entre comillas.
        tag_nombre, formato_cadena = token.split_contents()
    except ValueError:
        msg = '%r la etiqueta requiere un simple argumento' % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)
    return NodoFechaActual(formato_cadena[1:-1])
```

Hay muchas cosas en juego aquí:

- ***parser*** es la instancia del *parser*. No lo necesitamos en este ejemplo.
- ***token.contents*** es un *cadena* con los contenidos crudos de la etiqueta, en nuestro ejemplo sería: ‘fecha_actual “%Y-%m-%d %I:%M %p”’.
- El método token.split_contents() separa los argumentos en sus espacios, mientras deja unidas a las *cadenas*. Evite utilizar token.contents.split() (el cual usa la semántica natural de Python para dividir *cadenas*, y por esto no es tan robusto, ya que divide en todos los espacios, incluyendo aquellos dentro de cadenas entre comillas.
- Esta función es la responsable de generar la excepción django.template.TemplateSyntaxError con mensajes útiles, ante cualquier caso de error de sintaxis.

- No escribas el nombre de la etiqueta en el mensaje de error, ya que eso acoplaría innecesariamente el nombre de la etiqueta a la función. En cambio, *token.split_contents()[0]* siempre contendrá el nombre de tu etiqueta – aún cuando la etiqueta no lleve argumentos.

- La función devuelve NodoFechaActual (el cual mostraremos en un momento) conteniendo todo lo que el nodo necesita saber sobre esta etiqueta. En este caso, sólo pasa el argumento "%Y-%m-%d %I:%M %p". Las comillas son removidas con format_string[1:-1].

- Las funciones de compilación de etiquetas de plantilla *deben* devolver una subclase de *Node*; cualquier otro valor es un error.

Escribir el nodo de plantilla

El segundo paso para escribir etiquetas propias, es definir una subclase de Node que posea un método render(). Continuando con el ejemplo previo, debemos definir CurrentTimeNode:
import datetime

```
import datetime

class NodoFechaActual(template.Node):
    def __init__(self, formato_cadena):
        self.formato_cadena = str(formato_cadena)

    def render(self, context):
        ahora = datetime.datetime.now()
        return ahora.strftime(self.formato_cadena)
```

Estas dos funciones (__init__ y render) se relacionan directamente con los dos pasos para el proceso de la plantilla (compilación y renderizado). La función de inicialización sólo necesitará almacenar el string con el formato deseado, el trabajo real sucede dentro de la función render()

Del mismo modo que los filtros de plantilla, estas funciones de renderización deberían fallar silenciosamente en lugar de generar errores. En el único momento en el cual se le es permitido a las etiquetas de plantilla generar errores es en tiempo de compilación.

Registrar la etiqueta

Finalmente, deberás registrar la etiqueta con tu objeto Library dentro del módulo. Registrar nuevas etiquetas es muy similar a registrar nuevos filtros (como explicamos previamente). Sólo deberás instanciar un objeto template.Library y llamar a su método tag(). Por ejemplo:

```
register.tag('fecha_actual', fecha_actual)
```

El método *tag()* toma dos argumentos:

1. El nombre de la etiqueta de plantilla (*string*). Si esto se omite, se utilizará el nombre de la función de compilación.

2. La función de compilación.

De manera similar a como sucede con el registro de filtros, también es posible utilizar register.tag como un decorador en Python 2.7 o posterior:

```
@register.tag(name="fecha_actual")
def fecha_actual(parser, token):
    # ...

@register.tag
def fecha_actual(parser, token):
    # ...
```

Si omitimos el argumento name, así como en el segundo ejemplo, Django usará el nombre de la función como nombre de la etiqueta.

Definir una variable en el contexto

El ejemplo en la sección anterior simplemente devuelve un valor. Muchas veces es útil definir variables de plantilla en vez de simplemente devolver valores. De esta manera, los autores de plantillas podrán directamente utilizar las variables que esta etiqueta defina.

Para definir una variable en el contexto, asignaremos a nuestro objeto context disponible en el método render() nuestras variables, como si de un diccionario se tratase. Aquí mostramos la versión actualizada de NodoFechaActual que define una variable de plantilla, fecha_actual, en lugar de devolverla:

```
class NodoFechaActual2(template.Node):
    def __init__(self, formato_cadena):
        self.formato_cadena = str(formato_cadena)

    def render(self, context):
        ahora = datetime.datetime.now()
        context['fecha_actual'] = ahora.strftime(self.formato_cadena)
        return ''
```

Devolvemos una cadena vacía, debido a que render() siempre debe devolver un string. Entonces, si todo lo que la etiqueta hace es definir una variable, render() debe al menos devolver una cadena vacía.

De esta manera usaríamos esta nueva versión de nuestra etiqueta:

```
{% fecha_actual2 "%Y-%M-%d %I:%M %p" %}
<p>Fecha:{{ fecha_actual }}.</p>
```

Pero hay un problema con NodoFechaActual2: el nombre de la variable fecha_actual está definido dentro del código. Esto significa que tendrás que asegurar que {{ fecha_actual }} no sea utilizado en otro lugar dentro de la plantilla, ya que {% fecha_actual %} sobreescribirá el valor de esa otra variable.

Una solución más limpia, es recibiendo el nombre de la variable en la etiqueta de plantilla así:

```
{% traer_fecha_actual "%Y-%M-%d %I:%M %p" as mi_fecha_actual %}
<p>Fecha: {{ mi_fecha_actual }}.</p>
```

Para hacer esto, necesitaremos modificar tanto la función de compilación como la clase Node de la siguiente forma:

templatetags/etiquetas.py

```
import datetime
import re
from django import template

register = template.Library()

class NodoFechaActual3(template.Node):
    def __init__(self, formato_cadena, var_nombre):
        self.formato_cadena = str(formato_cadena)
        self.var_nombre = var_nombre

    def render(self, context):
        ahora = datetime.datetime.now()
        context[self.var_nombre] = ahora.strftime(self.formato_cadena)
        return ''

@register.tag(name="traer_fecha_actual")
def traer_hora_actual(parser, token):
    # Esta versión usa expresiones regulares para analizar el contenido de la etiqueta.
    try:
        # Dividir por None == dividir por espacios.
        tag_nombre, arg = token.contents.split(None, 1)
    except ValueError:
        msg = '%r La etiqueta requiere un simple argumento' % token.contents[0]
        raise template.TemplateSyntaxError(msg)

    m = re.search(r'(.*?) as (\w+)', arg)
    if m:
        fmt, var_nombre = m.groups()
    else:
        msg = '%r Argumentos no validos para la etiqueta' % tag_nombre
        raise template.TemplateSyntaxError(msg)
     if not (fmt[0] == fmt[-1] and fmt[0] in ("'", '"')):
        msg = "%r Los argumentos deben de ir entre comillas" % tag_nombre
        raise template.TemplateSyntaxError(msg)

    return NodoFechaActual3(fmt[1:-1], var_nombre)
```

Ahora, traer_fecha_actual() pasa la cadena de formato junto al nombre de la variable a NodoFechaActual3.

Evaluar hasta otra etiqueta de bloque

Las etiquetas de plantilla pueden funcionar como bloques que contienen otras etiquetas (piensa en {% if %}, {% for %}, etc.). Para crear una etiqueta como esta, usa parser.parse() en tu función de compilación.

Aquí vemos como está implementada una etiqueta {% coment %}:

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

parser.parse() toma una tupla de nombres de etiquetas de bloque para evaluar y devuelve una instancia de django.template.NodeList, la cual es una lista de todos los objetos Nodo que el *parser* encontró *antes* de haber encontrado alguna de las etiquetas nombradas en la tupla.

Entonces, en el ejemplo previo, nodelist es una lista con todos los nodos entre {% comment %} y {% endcomment %}, excluyendo a los mismos {% comment %} y {% endcomment %}.

Luego de que parser.parse() es llamado el *parser* aún no ha "consumido" la etiqueta {% endcomment %}, es por eso que en el código se necesita llamar explícitamente a parser.delete_first_token() para prevenir que esta etiqueta sea procesada nuevamente.

Luego, CommentNode.render() simplemente devuelve un *string* vacío. Cualquier cosa entre {% comment %} y {% endcomment %} es ignorada.

Evaluar hasta otra etiqueta de bloque y guardar el contenido

En el ejemplo anterior, do_comment() desechó todo entre {% comment %} y {% endcomment %}, pero también es posible hacer algo con el código entre estas etiquetas.

Por ejemplo, presentamos una etiqueta de plantilla, {% upper %}, que convertirá a mayúsculas todo hasta la etiqueta {% endupper %}:

```
{% upper %}
  Esto aparecerá en mayúsculas, {{tu_nombre }}.
{% endupper %}
```

Como en el ejemplo previo, utilizaremos parser.parse() pero esta vez pasamos el resultado de nodelist a Node así:

```
@register.tag
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):

    def __init__(self, nodelist):
        self.nodelist = nodelist

    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

El único concepto nuevo aquí es self.nodelist.render(context) en UpperNode.render(). El mismo simplemente llama a render() en cada Node en la lista de nodos.

Para más ejemplos de renderizado complejos, examina el código fuente para las etiquetas {% if %}, {% for %}, {% ifequal %} y {% ifchanged %}. Puedes encontrarlas en el directorio django/template/defaulttags.py.

Un atajo para etiquetas simples

Muchas etiquetas de plantilla reciben un único argumento–una cadena o una referencia a una variable de plantilla y retornan una cadena luego de hacer algún procesamiento basado solamente en el argumento de entrada e información externa. Por ejemplo la etiqueta fecha_actual que escribimos antes es de este tipo. Le pasamos una cadena de formato, y retorna la hora como una cadena.

Para facilitar la creación de esos tipos de etiquetas, Django provee una función auxiliar: simple_tag. Esta función, que es un método de django.template.Library, recibe una función que acepta un argumento, lo encapsula en una función render y el resto de las piezas necesarias que mencionamos previamente y lo registra con el sistema de plantillas.

Nuestra función fecha_actual podría entonces ser escrita de la siguiente manera:

```
def fecha_actual(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(fecha_actual)
```

Y la podemos registrar mediante un decorador así:

```
@register.simple_tag
def fecha_actual(token):
    ...
```

Un par de cosas a tener en cuenta acerca de la función auxiliar simple_tag:

- Sólo se pasa un argumento a nuestra función.
- La verificación de la cantidad requerida de argumentos ya ha sido realizada para el momento en el que nuestra función es llamada, de manera que no es necesario que lo hagamos nosotros.
- Las comillas alrededor del argumento (si existieran) ya han sido quitadas, de manera que recibimos una cadena común.

Etiquetas de inclusión

Otro tipo de etiquetas de plantilla común es aquel que visualiza ciertos datos renderizando *otra* plantilla. Por ejemplo la interfaz de administración de Django usa etiquetas de plantillas personalizadas (*custom*) para visualizar los botones en la parte inferior de la páginas de formularios "agregar/cambiar". Dichos botones siempre se ven igual, pero el destino del enlace cambia dependiendo del objeto que se está modificando. Se trata de un caso perfecto para el uso de una pequeña plantilla que es llenada con detalles del objeto actual.

Ese tipo de etiquetas reciben el nombre de *etiquetas de inclusión*. Es probablemente mejor demostrar cómo escribir una usando un ejemplo. Escribamos una etiqueta que produzca una lista de libros para un simple objeto Libro. Usaremos una etiqueta como esta:

```
{% libros_por_autor autor %}
```

El resultado será algo como esto:

```
<ul>
  <li>Libro uno</li>
  <li>Libro dos</li>
  <li>Otro libro</li>
</ul>
```

Primero definimos la función que toma un argumento y produce un diccionario de datos con los resultados. Nota que nos basta un diccionario y no necesitamos retornar nada más complejo.

Esto será usado como el contexto para el fragmento de plantilla:

```
def libros_por_autor(autor):
  libros = Libro.objects.filter(autores__id=autor.id)
  return {'libros': libros}
```

Luego creamos la plantilla usada para renderizar la salida de la etiqueta:

```
<ul>
{% for libro in libros %}
  <li>{{ libro.titulo }}</li>
{% endfor %}
</ul>
```

Finalmente una vez que la hemos creado, es necesario registrar la etiqueta de inclusión invocando al método *inclusion_tag()* sobre un objeto Library.

Continuando con nuestro ejemplo, si la plantilla se encuentra en un archivo llamado libros_por_autor.html, registraremos la plantilla de la siguiente manera:
register.inclusion_tag('libros_por_autor.html')(libros_por_autor)

Como siempre, la sintaxis de decoradores de Python también funciona, de manera que podemos haber escrito:

```
@register.inclusion_tag('libros_por_autor.html')
def libros_por_autor(autor):
  # ...
```

A veces tus etiquetas de inclusión necesitan tener acceso a valores del contexto de la plantilla padre. Para resolver esto Django provee una opción takes_context para las etiquetas de inclusión. Si especificas takes_context cuando creas una etiqueta de plantilla, la misma no tendrá argumentos obligatorios y la función Python subyacente tendrá un argumento: el contexto de la plantilla en el estado en el que se encontraba cuando la etiqueta fue invocada.

Por ejemplo supongamos que estás escribiendo una etiqueta de inclusión que será siempre usada en un contexto que contiene variables home_link y home_title que apuntan a la página principal.

Así es como se vería la función Python:

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

■**Nota:** El primer parámetro de la función *debe* llamarse *context.*

La plantilla *link.html* podría contener lo siguiente:

```
Ir directamente a <a href="{{ link }}">{{ title }}</a>.
```

Entonces, cada vez que desees usar esa etiqueta personalizada, carga su biblioteca y ejecútala sin argumentos, de la siguiente manera:

```
{% jump_link %}
```

Escribir cargadores de plantillas personalizados

Los cargadores de plantillas incluidos con Django (descritos en la sección "Etiquetas de inclusión" más arriba) cubrirán usualmente todas tus necesidades de carga de plantillas, pero es muy sencillo escribir el tuyo propio si necesitas alguna lógica especial en dicha carga. Por ejemplo podrías cargar plantillas desde una base de datos, o directamente desde un repositorio Subversion usando las librerías (*bindings*) Python de Subversion, o (como veremos) desde un archivo ZIP.

Un cargador de plantillas –esto es, cada entrada en la variables de configuración TEMPLATE_LOADERS– debe ser un objeto invocable (*callable*) con la siguiente interfaz: load_template_source(template_nombre, template_dirs=None)

El argumento template_nombre es el nombre de la plantilla a cargar (tal como fue pasado a loader.get_template() o loader.select_template()) y template_dirs es una lista opcional de directorios en los que se buscará en lugar de TEMPLATE_DIRS.

Si un cargador es capaz de cargar en forma exitosa una plantilla, debe retornar una tupla: (template_source, template_path). Donde template_source es la cadena de plantilla que será compilada por la maquinaria de plantillas, y template_path es la ruta desde la cual fue cargada la plantilla. Dicha ruta podría ser presentada al usuario para fines de depuración así que debe identificar en forma rápida desde dónde fue cargada la plantilla.

Si al cargador no le es posible cargar una plantilla, debe lanzar django.template.TemplateDoesNotExist.

Cada función del cargador debe también poseer un atributo de función is_usable. Este es un Booleano que le informa a la maquinaria de plantillas si este cargador está disponible en la instalación de Python actual. Por ejemplo el cargador desde eggs (que es capaz de cargar plantillas desde eggs Python) fija is_usable a False si el módulo pkg_resources no se encuentra instalado, porque pkg_resources es necesario para leer datos desde eggs.

Un ejemplo ayudará a clarificar todo esto. Aquí tenemos una función cargadora de plantillas que puede cargar plantillas desde un archivo ZIP. Usa una variable de configuración personalizada TEMPLATE_ZIP_FILES como una ruta de búsqueda en

lugar de TEMPLATE_DIRS y espera que cada ítem en dicha ruta sea un archivo ZIP que contiene plantillas:

```
import zipfile
from django.conf import settings
from django.template import TemplateDoesNotExist

def load_template_source(template_nombre, template_dirs=None):
    """Cargador de plantillas desde archivos ZIP."""
    template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])
    # Carga cada archivo ZIP de TEMPLATE_ZIP_FILES.
    for fnombre in template_zipfiles:
        try:
            z = zipfile.ZipFile(fnombre)
            source = z.read(template_nombre)
        except (IOError, KeyError):
            continue
        z.close()
        # Encuentra las plantillas y retorna el codigo.
        template_path = "%s:%s" % (fnombre, template_nombre)
        return (source, template_path)
    # Si en este punto la plantilla no ha sido cargada, lanzamos un error.
    raise  TemplateDoesNotExist(template_nombre)
# Este cargador siempre es usable (ya que zipfile está incluido en Python)
load_template_source.is_usable = True
```

El único paso restante si deseamos usar este cargador es agregarlo a la variable de configuración TEMPLATE_LOADERS. Si pusiéramos este código en un paquete llamado *misitio.zip_loader* entonces agregariamos *misitio.zip_loader.load_template_source* a TEMPLATE_LOADERS.

Usar la referencia de plantillas incorporadas

La interfaz de administración de Django incluye una referencia completa de todas las etiquetas y filtros de plantillas disponibles para un sitio determinado. Está designada para ser una herramienta que los programadores Django proveen a los desarrolladores de plantillas. Para activarla sigue los siguientes pasos:

- Agrega la aplicación 'django.contrib.admindocs' al archivo settings.py en la variable INSTALLED_APPS (como lo harías con cualquier otra aplicación).
- Agrega el patrón, (r'^admin/doc/', include('django.contrib.admindocs.urls')) al archivo urls.py, solo asegúrate que aparezca antes que el patrón que apunta a la interfaz administrativa (r'^admin/')
- Dirige tu navegador a: /admin/doc/.

Si entras a la interfaz de administración, da clic en el nuevo enlace Documentación en la zona superior derecha de la página.

La referencia está dividida en cuatro secciones: etiquetas, filtros, modelos y vistas. Las secciones *etiquetas* y *filtros* describen todas las etiquetas incluidas (en efecto, las referencias de etiquetas y filtros del *capítulo 4* han sido extraídas directamente de

esas páginas) así como cualquier biblioteca de etiquetas o filtros personalizados disponible.

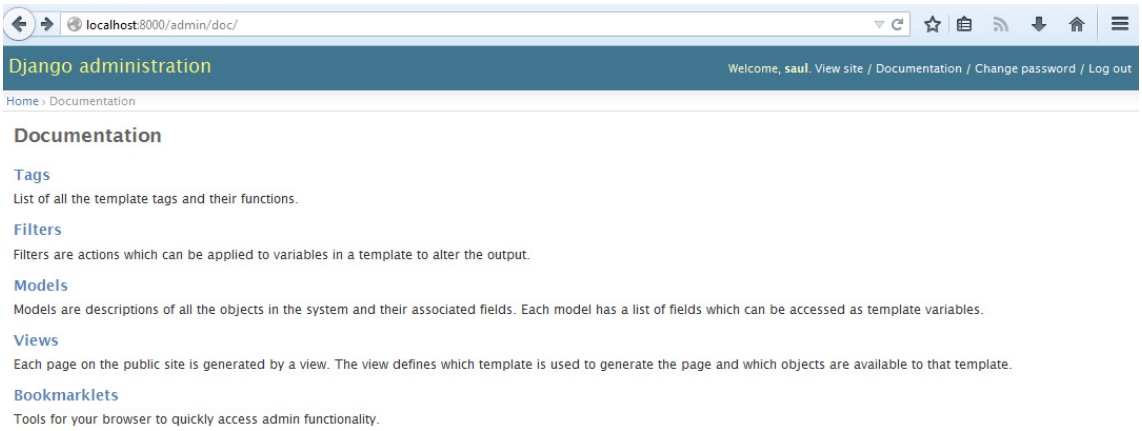


Imagen 9.1 El framework de documentación de Django

La página *views* es la más valiosa. Cada URL en tu sitio tiene allí una entrada separada. Si la vista relacionada incluye una cadena de documentación o docstring, haciendo clic en la URL te mostrará lo siguiente:

- El nombre de la función de vista que genera esa vista.
- Una breve descripción de lo qué hace la vista.
- El contexto, o una lista de variables disponibles en la plantilla de la vista.
- El nombre de la plantilla o plantillas usados para esa vista.

Para un ejemplo detallado de la documentación de vistas, lee el código fuente de la vista genérica de Django object_list la cual se encuentra en django/views/generic/list_detail.py.

Debido a que los sitios implementados con Django generalmente usan objetos de bases de datos, las páginas *models* describen cada tipo de objeto en el sistema así como todos los campos disponibles en esos objetos.

En forma conjunta, las páginas de documentación deberían proveerte cada etiqueta, filtro, variable y objeto disponible para su uso en una plantilla arbitraria.

Configurando el sistema de plantillas en modo autónomo

■**Nota:** Estas sección es sólo de interés para aquellos que intentan usar el sistema de plantillas como un componente de salida en otra aplicación. Si estás usando el sistema como parte de una aplicación Django, la información aquí presentada no es relevante para ti.

Normalmente Django carga toda la información de configuración que necesita desde su propio archivo de configuración por omisión, combinado con las variables de configuración en el módulo indicado en la variable de entorno DJANGO_SETTINGS_MODULE. Pero si estás usando el sistema de plantillas

independientemente del resto de Django, el esquema de la variable de entorno no es muy conveniente porque probablemente quieras configurar el sistema de plantillas en una manera más acorde con el resto de tu aplicación en lugar de tener que vértelas con archivos de configuración e indicando los mismos con variables de entorno.

Para resolver este problema necesitas usar la opción de configuración manual descrita en forma completa en el Apéndice E. En resumen, necesitas importar las partes apropiadas del sistema de plantillas y entonces, *antes* de invocar alguna de las funciones de plantillas, invoca a django.conf.settings.configure() con cualquier valor de configuración que desees especificar.

Podrías desear considerar fijar al menos TEMPLATE_DIRS (si vas a usar cargadores de plantillas), DEFAULT_CHARSET (aunque el valor por omisión utf-8 probablemente sea adecuado) y TEMPLATE_DEBUG. Todas las variables de configuración están descritas en el Apéndice E y todas las variables cuyos nombres comienzan con TEMPLATE son de obvio interés.

¿Qué sigue?

Continuamos esta sección con temas avanzados, el *siguiente capítulo* trata sobre el uso avanzado de los modelos en Django.

CAPÍTULO 10

Modelos avanzados

En el capítulo 5, presentamos una pequeña introducción a la capa de base de datos – la cual define los modelos de la base de datos y la forma de usar la API para crear, recuperar, actualizar y borrar registros. En este capítulo, nos adentraremos un poco más, en las características avanzadas de la capa de modelos de Django.

Relaciones entre objetos

Recuerdas el modelo que creamos en el *capítulo 5*:

libros/models.py

```
from django.db import models

class Editor(models.Model):
    nombre = models.CharField(max_length=30)
    direccion = models.CharField(max_length=5)
    ciudad = models.CharField(max_length=60)
    estado = models.CharField(max_length=30)
    pais = models.CharField(max_length=30)
    website = models.URLField()

    class Meta:
        ordering = ["nombre"]
        verbose_name_plural = "Editores"

    def __str__(self):              # __unicode__ en Python 2
        return self.nombre

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField('e-mail', blank=True)

    def __str__(self):                  # __unicode__ en Python 2
        return '%s %s' % (self.nombre, apellidos)

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editor = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField(blank=True, null=True)
    portada = models.ImageField(upload_to='portadas')

    class Meta:
        ordering = ["nombre"]
```

```
    verbose_name_plural = "Autores"

def __str__(self):              # __unicode__ en Python 2
    return self.titulo
```

Como explicamos en el *capítulo 5*, para acceder directamente al valor de un determinado campo en la base de datos, es necesario usar los atributos del objeto. Por ejemplo, para determinar el titulo de un libro con un ID 5 (ID: El identificador o clave primaria, siempre que esta exista), hacemos lo siguiente:

```
>>> from biblioteca.models import Libro
>>> b = Libro.objects.get(id=5)
>>> b.titulo
'The definitive guide'
```

Pero una cosa que no mencionamos previamente, fueron las relaciones entre objetos – Nos referimos a los campos expresados como ForeignKey o ManyToManyField, los cuales actúan de modo ligeramente diferente.

Accediendo a valores en claves foráneas

Cuando accedes a un campo del tipo ForeignKey (Relación foránea, del tipo muchos a uno)

Puedes obtener el modelo del objeto relacionado, de la siguiente forma:

```
>>> from biblioteca.models import Editor, Libro
>>> b = Libro.objects.get(id=5)
>>> b.editor
<Publisher: Apress Publishing>
>>> b.editor.website
u'http://www.apress.com/'
```

Como habrás notado, la forma en que trabajan los campos ForeignKey, es ligeramente diferente, debido a la naturaleza poco simétrica de la relación. Por ejemplo para conseguir una lista de libros de un editor en especifico, usa editores.libro_set.all(), así:

```
>>> p = Editor.objects.get(nombre='Apress Publishing')
>>> p.libro_set.all()
[<Libro: The Django Libro>, <Libro: Dive Into Python>, ...]
```

Detrás de escena, libro_set es solo un QuerySet (el cual cubrimos en el Capítulo5) y puede ser filtrado y rebanado, tal como cualquier otro QuerySet. Por ejemplo:

```
>>> p = Editor.objects.get(nombre='Apress Publishing')
>>> p.libro_set.filter(titulo__icontains='django')
[<Libro: The definitive guide>, <Libro: Pro Django>]
```

El nombre del atributo libro_set es generado agregando el nombre del modelo en minúsculas a _set.

Accediendo a valores en claves muchos a muchos

Los valores muchos a muchos (Many-to-many) trabajan de forma parecida a los valores en campos foráneos, a menos que tratemos con valores QuerySet en lugar de instancias del modelo. Por ejemplo, este es la forma que se muestran los autores de un libro:

```
>>> b = Libro.objects.get(id=5)
>>> b.autores.all()
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]
>>> b.autores.filter(nombre='Adrian')
[<Author: Adrian Holovaty>]
>>> b.autores.filter(nombre='Adam')
[]
```

Esto también trabaja en orden inverso, Para visualizar todos los libros de un determinado autor, usa autor.libro_set, así:

```
>>> from biblioteca.models import Autor
>>> a = Autor.objects.get(nombre='Adrian', apellidos='Holovaty')
>>> a.libro_set.all()
[<Libro: The Django Libro>, <Libro: Adrian's Other Libro>]
```

Al igual que con lo campos f*oráneos,* el nombre del atributo *libro_set* es generado agregando el nombre del modelo en minúsculas a _set.

Como realizar cambios al esquema de la base de datos

Cuando introducimos el comando migrate en el capítulo 5, hicimos énfasis en que el comando migrate crea las tablas que no existen en la base de datos, sincronizando los modelos – pero nos falto comentar que no solo crea modelos, también agrega y renombra campos, elimina y agrega modelos, por lo que en la mayoría de casos no necesitaras hacer estos cambios manualmente.

Esta sección explica cómo hacer cambios al esquema de la base de datos, usando los comandos migrate y makemigrations.

Cuando se trata con cambios al esquema de la base de datos, es importante tener en cuenta algunas cuestiones relacionadas con la forma en que trabaja la capa de base de datos de Django:

- Django se quejará estrepitosamente si un modelo contiene un campo que todavía no se ha creado en la tabla de base de datos. Esto causará un error la primera vez que utilices la API de la base de datos (es decir, esto sucede en tiempos de ejecución del código, no en tiempos de compilación).

- A Django no le importa si una tabla de la base de datos, contiene columnas que no están definidas en el modelo.

- A Django no le importa si una base de datos contiene una tabla que no está representada por un modelo.

Por lo que realizar cambios al esquema, solo es cuestión de cambiar varias piezas – el código Python y la base de datos en sí mismo en el orden correcto.

Migraciones

Los dos comandos que se encargan de interactuar con las migraciones y manejar el esquema de la base de datos en Django son:

makemigrations:	Comando responsable de crear nuevas migraciones basadas en los cambios hechos a los modelos.
migrate:	Comando responsable de aplicar las migraciones y sincronizar los modelos de la base de datos y listar su estatus.

Las migraciones son la forma en que Django propaga los cambios que realizas a tus modelos (agregando campos, eliminando un modelo, etc.) en el esquema de la base de datos. Se diseñan para ser sobre todo automáticas, pero necesitas tener cuidado con los problemas comunes que puedan surgir, ya que un error puede dejar inutilizable tu base de datos.

Un poco de historia

Antes de la versión 1.7 de Django, solo era posible agregar modelos nuevos a la base de datos; no era posible alterar o quitar los modelos existentes, usando el comando syncdb (el precursor de migrate). Por lo que si se agregaba y cambiaba un campo de un modelo, o si se eliminaba un modelo, era necesario realizar el cambio en la base de datos manualmente usando SQL.

Por lo que surgieron algunas herramientas de terceros en especial *South* que proveían soporte para realizar estos cambios adicionales en el esquema de la base de datos, con el tiempo este tipo de soporte fue considerado, lo suficientemente importante que fue incluido en el código base de Django.

El sistema de migraciones de Django, se divide en dos partes: la lógica que calcula y almacena las operaciones que se ejecutan (django.db.migrations), y la capa de abstracción de la base de datos que se encarga de cosas como "crear un modelo" "borrar un campo" en SQL –el cual es trabajo del editor de esquema (SchemaEditor).

Trabajar con migraciones es sencillo, cambia tu modelo – es decir agrega un campo y/o remueve un modelo... – y ejecuta los comandos **makemigrations** y **migrate** en ese orden.

Agregar campos

Cuando se agrega un campo a una tabla/modelo en un entorno en producción, el truco es realizarlo primero en un servidor de desarrollo y luego llevar los cambios al sitio de producción.

Sin embargo las migraciones funcionarán de la misma manera en el mismo conjunto de datos y producirán resultados constantes, lo que significa que pueden ser usadas en desarrollo y en producción bajo las mismas condiciones y circunstancias y con los mismos resultados.

Para agregar campos a un modelo sigue estos pasos:

1. Agrega el campo a tu modelo.
2. Asegúrate que el campo incluya las opciones blank=True o null=True (si es un campo basado en fechas o numérico).
3. Ejecuta el comando manage.py makemigrations, para grabar los cambios.
4. Sincroniza los modelos con manage.py migrate.

5. Ejecuta manage.py shell y verifica que el nuevo campo se haya agregado correctamente, importa el modelo y realiza una consulta a la base de datos (por ejemplo con Libro.objects.all()[:5]). Si la actualización fue correcta, la declaración anterior debe trabajar sin errores.

Como ejemplo, examinemos los pasos necesarios para agregar un campo **num_paginas** al modelo Libro del *capítulo 5.*

Lo primero que haremos será cambiar el modelo y agregarle el nuevo campo así:

```
class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editor = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField(blank=True, null=True)
    portada = models.ImageField(upload_to='portadas')
    num_paginas = models.IntegerField(blank=True, null=True)
```

■**Nota:** Lee la sección "Crear campos opcionales" del *capítulo 6*, para conocer los detalles importantes acerca del porque incluimos blank=True y null=True.

Una vez que hemos cambiado nuestro modelo, podemos validarlo usando el comando python manage.py check, para asegurarnos de que todo está correctamente en su lugar, si el comando no lanza ningún error, podemos crear las migraciones con el comando python manage.py makemigrations, con el que obtendremos una salida como esta:

```
Migrations for 'biblioteca':
  0002_auto_20150216_1318.py:
    -Add field num_paginas to libro
```

Tus modelos serán explorados y comparados con las versiones contenidas en el fichero actual de migraciones, y entonces un nuevo conjunto de migraciones será escrito. Asegúrate de leer la salida para ver lo qué el comando makemigrations piensa que ha cambiado – no es perfecto, sobre todo en cambios complejos, puede ser que no detecte lo que esperabas.

Ahora podemos ejecutar el comando python manage.py sqlmigrate biblioteca para ver la declaración CREATE TABLE de SQL. Dependiendo de la base de datos la salida puede variar:

```
BEGIN;
CREATE TABLE "biblioteca_libro_new" (
    "id" serial NOT NULL PRIMARY KEY,
    "titulo" varchar(100) NOT NULL,
    "editores_id" integer NOT NULL REFERENCES "libros_editores" ("id"),
    "fecha_publicacion" date NOT NULL,
    "num_paginas" integer NULL
    #...
);
```

La nueva columna es representada en SQL así:

```
"num_pages"  integer NULL
```

¿POR QUÉ AGREGAMOS COLUMNAS NOT NULL?

En este punto vale la pena hacer mención de una delicada sutileza, cuando agregamos el campo num_paginas al modelo, incluimos las opciones blank=True y null=True ya que una columna de una base de datos, debe de contener valores NULL cuando es creada por primera vez.

Sin embargo también es posible agregar columnas que no contengan valores NULL. Para ello es necesario crear las columnas como NULL y luego agregar valores por defecto y después alterar la columna para modificarla a NOT NULL.

Usando SQL tendríamos que hacer esto:

```
BEGIN;
ALTER TABLE libros_libro ADD COLUMN num_paginas integer;
UPDATE libros_libro SET num_pages=0;
ALTER TABLE libros_libro ALTER COLUMN num_paginas SET NOT NULL;
COMMIT;
```

Afortunadamente el comando migrate se encarga de realizar este trabajo por ti, ya que detecta cuando se quiera agregar un campo que contiene valores NOT NULL, advirtiéndonos que tratamos de agregar un campo NOT NULL y que no se puede hacer eso (ya que la base de datos necesita rellenar las filas existentes.)

Por lo que interactivamente nos ofrece dos opciones para corregirlo:

1. Introducir un valor por defecto, para rellenar las filas.

2. Salir para corregir el valor por defecto en los modelos.

Si seleccionamos la primera opción, se inicia el intérprete interactivo que nos pide agregar un valor por defecto, siempre y cuando sea un valor valido en Python, la segunda opción es obvia, es tu responsabilidad corregir el modelo manualmente usando SQL.

Una vez que tengas un nuevo archivo de migraciones, debes aplicarlo a la base de datos usando el comando `python manage.py migrate` para cerciorarte de que trabaja según lo previsto:

```
Operations to perform:
  Synchronize unmigrated apps: sessions, admin, messages, auth,
staticfiles, contenttypes
  Apply all migrations: biblioteca
Synchronizing apps without migrations:
  Creating tables...
  Installing custom SQL...
  Installing indexes...
Installed 0 object(s) from 0 fixture(s)
Running migrations:
  Applying libros.0002_auto... OK
```

El comando se ejecuta en dos etapas, primero sincroniza las aplicaciones que no han sido migradas (sincronizando los cambios en los modelos) y segundo ejecuta las migraciones que no han sido aplicadas. Una vez que una migración ha sido aplicada, esta se guarda en el sistema de control de versiones.

Una vez realizados los cambios inicia el intérprete interactivo, y comprueba que todo trabaje según lo planeado:

```
>>>from biblioteca.models import Libro
>>>Libro.objects.all()[:5]
>>>
```

Eliminar campos

Eliminar un campo de un modelo es mucho más fácil que agregar uno. Para remover un campo, solo sigue estos pasos:

1. Remueve el campo de tu modelo.

2. Ejecuta el comando python manage.py makemigrations, para grabar los cambios.

3. Haz los cambios en la base de datos con el comando python manage.py migrate.

4. Y reinicia el servidor Web.

Asegúrate de hacerlo en este orden.

Eliminar relaciones muchos a muchos

Debido a que los campos muchos a muchos son diferentes a los campos normales, el proceso de eliminación es un poco diferente.

Para eliminar manualmente una relación muchos a muchos, tendríamos que hacer lo siguiente:

1. Remueve el campo *ManyToManyField* de tu modelo y reinicia el servidor Web.

2. Remueve la tabla muchos a muchos de tu base de datos usando un comando SQL como este:
   DROP TABLE libros_libro_autores;

Como en la sección anterior, podemos seguir los siguientes pasos, para usar las migraciones:

1. Remueve el campo muchos a muchos de tu modelo.

2. Ejecuta el comando python manage.py makemigrations, para grabar los cambios.

3. Haz los cambios en la base de datos con el comando python manage.py migrate.

4. Reinicia el servidor Web.

Asegúrate de hacerlo en este orden y asegúrate de borrar cualquier modelo que dependa de la relación muchos a muchos, de otra forma no funcionara.

Eliminar modelos

Eliminar un modelo completo es tan fácil como agregar uno. Para remover un modelo, solo sigue estos pasos:

1. Remueve el modelo.
2. Ejecuta el comando python manage.py makemigrations, para grabar los cambios.
3. Haz los cambios en la base de datos con el comando python manage.py migrate.
4. Reinicia el servidor Web.

Manualmente puedes hacerlo así:

1. Elimina el modelo de tu archivo models.py y reinicia el servidor.
2. Elimina la tabla de tu base de datos, usando el siguiente comando:

```
DROP TABLE biblioteca_libro;
```

☠ **Advertencia:** Ten en cuenta, que también es necesario remover cualquier tabla que dependa de el modelo que quieras borrar de tu base de datos primero –por ejemplos alguna tabla que contenga un campo *foráneo* o una relación *muchos a muchos.*

Cómo en las secciones anteriores. Asegúrate de hacerlo en este orden.

Manejadores o Managers

En la declaración Libro.objects.all(), objects es un atributo especial a través del cual se realizan las consultas a la base de datos. En el Capítulo *Capítulo5*, identificamos brevemente a este como el *Manejador* (Manager). Es hora de sumergirnos en las profundidades y conocer que son los manejadores y cómo podemos usarlos.

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

Nombres de Manager

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

```
from django.db import models

class Persona(models.Model):
...#
   gente = models.Manager()
```

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

Managers Personalizados

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

La forma en que trabajan las clases Manager está documentada en *apéndice B*. Esta sección trata específicamente las opciones del modelo que personaliza el comportamiento del Manager.

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

Agregando Métodos Extra al Manager

Agregar métodos extra al Manager es la forma preferida de agregar funcionalidad a "nivel de tabla" a tus modelos. (Para agregar funcionalidad a nivel de registro – esto es, funciones que actúan sobre una instancia simple de un objeto modelo – usa métodos en los modelos, los cuales son explicados más adelante en este capítulo.

Por ejemplo, démosle al Manager de nuestro modelo Libro un método contar_titulos(), el cual toma una palabra clave y retorna el número de libros que contienen un titulo que contiene dicha palabra. (Este ejemplo es bastante superficial, pero demuestra cómo trabajan los Managers)

```
# models.py
from django.db import models

# ... Los modelos Autor y Editor van aquí ...

class ManejadorLibros(models.Manager):
   def contar_titulos(self, keyword):
      return self.filter(titulo__icontains=keyword).count()

class Libro(models.Model):
   titulo = models.CharField(max_length=100)
   autores = models.ManyToManyField(Autor)
   editores = models.ForeignKey(Editor)
   fecha_publicacion = models.DateField(blank=True, null=True)
   num_paginas = models.IntegerField(blank=True, null=True)
   objects = ManejadorLibros()

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

Con el Manager en su lugar, ahora podemos hacer esto:

```
>>> Libro.objects.contar_titulos('django')
4
>>> Libro.objects.contar_titulos('python')
18
```

Algunas notas sobre el código:

- Creamos una clase ManejadorLibros que extiende a la clase django.db.models.Manager. El sencillo método contar_titulos(), hace los cálculos. Observa que el método usa un filtro self.filter(), donde self se refiere al manager en sí mismo.

- Asignamos ManejadorLibros() a los atributos de objects en el modelo. Esto tiene el efecto de reemplazar el manejado por "defecto", el cual es llamado objects y es automáticamente creado si no especificas un manager personalizado. Lo hemos llamado objects en vez de usar algún otro nombre, para ser consistentes con los managers automáticamente creados por Django.

¿Por qué agregamos un método tal como contar_titulos() aquí? Bueno, para encapsular la ejecución de consultas comunes, a fin de de evitar duplicar el código.

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

Por ejemplo, este Manager personalizado ofrece un método contar(), que retorna una lista de todos los objetos Opinión, cada uno con un atributo extra num_respuestas; que es el resultado de una consulta agregada:

```
from django.db import models

class ManagerEncuestas(models.Manager):
    def contar(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.pregunta, p.fecha, COUNT(*)
            FROM biblioteca_opinion p, biblioteca_respuesta r
            WHERE p.id = r.encuesta_id
            GROUP BY p.id, p.pregunta, p.fecha
            ORDER BY p.fecha DESC""")
        lista_resultados = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], pregunta=row[1], fecha=row[2])
            p.num_respuestas = row[3]
            lista_resultados.append(p)
        return lista_resultados

class Opinion(models.Model):
    pregunta = models.CharField(max_length=200)
    fecha = models.DateField()
    objects = ManagerEncuestas()

class Respuesta(models.Model):
    encuesta = models.ForeignKey(Opinion)
    nombre = models.CharField(max_length=50)
    respuesta = models.TextField()
```

En este ejemplo, puedes usar Opinion.objects.contar() para retornar la lista de objetos Opinion con el atributo num_respuestas.

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

Modificando los QuerySets iníciales del Manager

Un QuerySet base de un Manager devuelve todos los objetos en el sistema. Por ejemplo, Libro.objects.all() retornará todos los libros de la base de datos.

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

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

```
from django.db import models

# Primero, definimos una subclase para el Manager.
class DahlLibroManager(models.Manager):
    def get_query_set(self):
        return super(DahlLibroManager, self).get_query_set().filter(autor='Roald Dahl')

# Despues lo anclamos al modelo Libro explícitamente.
class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autor = models.CharField(max_length=5)
    # ...
    objects = models.Manager() # El manager predeterminado.
    dahl_objects = DahlLibroManager() # El manager "Dahl" especial.
```

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

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

```
Libro.dahl_objects.all()
Libro.dahl_objects.filter(titulo='Matilda')
Libro.dahl_objects.count()
```

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

Aquí hay un ejemplo:

```
class ManejadorHombres(models.Manager):

    def get_query_set(self):
        return super(ManejadorHombres, self).get_query_set().filter(sexo='H')
```

```
class ManejadorMujeres(models.Manager):

    def get_query_set(self):
        return super(ManejadorMujeres, self).get_query_set().filter(sexo='M')

class Persona(models.Model):
    nombre = models.CharField(max_length=50)
    apellido = models.CharField(max_length=50)
    sexo = models.CharField(max_length=1, choices=(('M', 'Mujer'), ('H', 'Hombre')))
    gente = models.Manager() # EL manejador predeterminado.
    hombre = ManejadorHombres()# Devuelve solo hombres.
    mujer = ManejadorMujeres()# Devuelve solo mujeres.
```

Este ejemplo te permite consultar Persona.hombre.all(), Persona.mujer.all(), y Persona.gente.all(), con los resultados predecibles.

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

En resumen, Un Manager es la interfaz a través de la cual se proveen las operaciones de consulta de la base de datos a los modelos de Django. Existe al menos un Manager para cada modelo en una aplicación Django. Puedes crear manejadores personalizados para modificar el acceso a la base de datos para requisitos particulares.

Métodos de un Modelo

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

Esta es una técnica valiosa para mantener la lógica del negocio en un sólo lugar: el modelo.

Por ejemplo, este modelo tiene algunos métodos personalizados:

```
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length=15)
    apellido = models.CharField(max_length=15)
    nacimiento = models.DateField()
    domicilio = models.CharField(max_length=100)
    ciudad = models.CharField(max_length=15)
    estado = models.CharField(max_length=2)

    def estatus_bebe(self):
        "Retorna el estatus baby-boomer de la persona."
        import datetime
         if datetime.date(1945, 8, 1) <= self.nacimiento <= datetime.date(1964, 12, 31):
```

```
            return "Baby boomer"
        if self.nacimiento < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        return "Post-boomer"

def es_del_medio_oeste(self):
    "Retorna True si la persona nacio en el medio-oeste."
    return self.estado in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')

def _nombre_completo(self):
     "Retorna el nombre completo de una persona."
     return '%s %s' % (self.nombre, self.
```

Este es un ejemplo de su uso:

```
>>> p = Persona.objects.get(nombre='Barack', apellido='Obama')
>>> p.nacimiento
datetime.date(1961, 8, 4)
>>> p.estatus_bebe()
'Baby boomer'
>>> p.es_del_medio_oeste()
True
>>> p.nombre_completo # Nota que no es un método -- es tratado como un atributo
u'Barack Obama'
```

Existe también un puñado de métodos de modelo que tienen un significado "especial" para Python o Django. Estos métodos se describen a continuación.

__unicode__

El método ***__unicode__()*** es un "método mágico" de Python que define lo que debe ser devuelto si llamas a unicode() sobre el objeto. Django usa unicode(obj) (o la función relacionada str(obj) que se describe más abajo) en varios lugares, particularmente como el valor mostrado para hacer el render de un objeto en el sitio de administración de Django y como el valor insertado en un plantilla cuando muestra un objeto. Por eso, siempre debes retornar un string agradable y legible por humanos en el formato __unicode__() de un objeto. A pesar de que esto no es requerido, es altamente recomendado, sobre todo usando Python2X.

Por ejemplo:

```
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length=50)
    apellido = models.CharField(max_length=50)

    def __unicode__(self):
        return u'%s %s' % (self.nombre, self.apellido)
```

__str__

__str__() es un "método mágico" de Python que define lo que debe ser devuelto si llamas a str(). En Python3, Django usa str(obj), en varios lugares, particularmente como el valor mostrado para hacer el render de un objeto en el sitio de administración de Django y como el valor insertado en un plantilla cuando muestra

un objeto. Por eso, siempre debes retornar un string agradable y legible por humanos en el __str__ de un objeto. A pesar de que esto no es requerido, es altamente recomendado, al usar Python3.

Aquí hay un ejemplo:

```
class Persona(models.Model):
    nombre = models.CharField(maxlength=50)
    apellido = models.CharField(maxlength=50)

    def __str__(self):
        return '%s %s' % (self.nombre, self.apellido)
```

__eq__

Un método de igualdad que es definido cuando las instancias con la misma clave primaria son evaluadas y la clase en concreto es considerada igual.

Por ejemplo:

```
from django.db import models

class MiModelo(models.Model):
    id = models.AutoField(primary_key=True)

class ModeloProxy(MiModelo):
    class Meta:
        proxy = True

class HerenciaMultiple(MiModelo):
    pass

MiModelo(id=1) == MiModelo(id=1)
MiModelo(id=1) == ModeloProxy(id=1)
MiModelo(id=1) != HerenciaMultiple(id=1)
MiModelo(id=1) != MiModelo(id=2)
```

__hash__

Calcula el valor hash para una clave primaria de cualquier campo.

El método __hash__ está basado en la instancia del valor de la clave primaria. Usando (obj.pk). Si la instancia no tiene una clave primaria lanzara un error TypeError.

get_absolute_url

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

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

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

Sin embargo, a pesar de que este código es simple, no es muy portable, por lo que la mejor forma de aprovechar esto es usando la función reverse()
Por ejemplo:

```
def get_absolute_url(self):
    from django.core.urlresolvers import reverse
    return reverse('gente.views.detalles', args=[str(self.id)])
```

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

☠**Advertencia:** Debes evitar en lo posible, construir URL para entradas no validas, para reducir las posibilidades de enlazar contenidos y redireccionamientos no deseados y potencialmente peligrosos, por ejemplo:

```
def get_absolute_url(self):
    return '/%s/' % self.nombre
```

Si self.nombre es '/example.com' este devolverá '//example.com/' el cual es un esquema URL valido, pero no esperado.

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

```
<!– MALO ¡No hagas esto! –> <a href=”/gente/{{ object.id }}/”>{{ object.nombre }}</a>
```

Pero este es bueno:

```
<a href=”{{ object.get_absolute_url }}”>{{ object.nombre }}</a>
```

La lógica aquí, es que puedes cambiar la estructura completa, de las URL de tus objetos, en un único lugar con get_absolute_url().

■**Nota: La** cadena que devuelve el método get_absolute_url() deve contener **únicamente** caracteres ASCII (requeridos por las especificaciones URI **RFC 2396**) y deben ser encodificadas de ser necesario (Django provee filtros para esto).
Puedes usar la función django.utils.encoding.iri_to_uri() para realizar este trabajo y así garantizar que tus cadenas únicamente contendrán caracteres dentro del rango ASCII.

Sobrescribir métodos predefinidos de un modelo

Existe otro conjunto de métodos en un modelo que encapsulan un montón de comportamientos de la base de datos, estos pueden ser sobrescritos para requisitos particulares que quieras modificar. Particularmente los métodos save() y delete().
Siente libre de sobrescribir estos métodos (y cualquier otro método de un modelo) para alterar su comportamiento.
Un uso clásico para sobrescribir los métodos incorporados de un modelo, es por ejemplo si quieres que pase algo cuando guardas un objeto, así:

```
from django.db import models

class Autor(models.Model):
   nombre = models.CharField(max_length=30)
   apellidos = models.CharField(max_length=40)
   email = models.EmailField()

    def save(self, *args, **kwargs):
      haz_algo()
      super(Autor, self).save(*args, **kwargs) # Llama al "verdadero" método save().
```

También puedes evitar guardar un objeto en específico:

```
from django.db import models

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField()

   def save(self, *args, **kwargs):
       if self.nombre == "Foo":
           return # ¡Foo ha dicho que nunca publicara un libro!
       else:
           super(Autor, self).save(*args, **kwargs) # Llama al "verdadero" método save()
```

Es importante recordar llamar el método de los superclase - que es en este caso: super(Autor, self).save(*args, **kwargs) para asegurarse de que el objeto se ha guardado en la base de datos, ya que el comportamiento por omisión no toca la base de datos y el objeto no se guardara.

Es también importante pasar los argumentos que se pueda necesitar pasar al método del modelo - que son *args, **kwargs. De vez en cuando en Django, se amplían las capacidades incorporados de los métodos de los modelos, agregando nuevos argumentos. Si utilizas *args y **kwargs en tus definiciones del método, garantizas que tu código soportará automáticamente esos argumentos cuando se agreguen mas.

Ejecutando consultas personalizadas en SQL

Algunas veces te encontraras con que la API de bases de datos de Django únicamente te permite realizar un cierto tipo de consultas, siéntete libre de escribir sentencias SQL personalizadas en métodos personalizados de modelo y métodos a nivel de módulo. Es muy sencilla accede al objeto django.db.connection el cual representa la conexión actual a la base de datos. Para usarla, invoca el método connection.cursor() para obtener un objeto cursor. Después, llama a cursor.execute(sql, [params]) para ejecutar sentencias SQL, y cursor.fetchone() o cursor.fetchall() para devolver las filas resultantes:

Por ejemplo:

```
 >>> from django.db import connection
>>> cursor = connection.cursor()
>>> cursor.execute("""
...    SELECT DISTINCT nombre
```

```
...     FROM personas
...     WHERE apellido = %s""", ['Lennon'])
>>> row = cursor.fetchone()
>>> print row
['John']
```

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

En vez de ensuciar el código de tu vista con esta declaración django.db.connection, es una buena idea ponerlo en un método personalizado en el modelo o en un método de un manager.

De esta forma, el anterior ejemplo puede ser integrado en un método de manager, así:

```
class PersonaManager(models.Manager):
    def nombres(self, apellido):
        cursor = connection.cursor()
        cursor.execute("""
            SELECT DISTINCT apellido
            FROM persona
            WHERE apellido = %s""", [apellido])
        return [row[0] for row in cursor.fetchone()]

class Persona(models.Model):
    nombre = models.CharField(max_length=15)
    apellido = models.CharField(max_length=15)
    objects = PersonaManager()
```

Y este es un ejemplo de su uso:

```
>>> Persona.objects.nombres('Lennon')
['John', 'Cynthia']
```

¿Que sigue?

En el siguiente capítulo, te mostraremos el framework “Vistas genéricas”, el cual te permite ahorrar tiempo para construir sitios Web, que siguen patrones comunes.

CAPÍTULO 11

Vistas Genéricas

De nuevo aparece aquí un tema recurrente en este libro: en el peor de los casos, el desarrollo Web es aburrido y monótono. Hasta aquí, hemos cubierto cómo Django trata de alejar parte de esa monotonía en las capas del modelo y las plantillas, pero los desarrolladores Web también experimentan este aburrimiento al nivel de las vistas.

Las **vistas genéricas basadas en clases** de Django fueron desarrolladas para aliviar ese dolor. Recogen ciertos estilos y patrones comunes encontrados en el desarrollo de vistas y los abstraen, de modo que puedas escribir rápidamente vistas comunes de datos sin que tengas que escribir mucho código. De hecho, casi todos los ejemplos de vistas en los capítulos precedentes pueden ser reescritos con la ayuda de vistas genéricas, usando clases.

Él *capítulo 8*, se refirió brevemente a la forma de crear una vista "genérica". Para repasar, podemos empezar por reconocer ciertas tareas comunes, como mostrar una lista de objetos, y escribir el código que muestra una lista de *detalle* de cualquier objeto. Por lo tanto el modelo en cuestión puede ser pasado como un argumento extra a la URLconf.

Django viene con vistas genéricas, basadas en clases para hacer lo siguiente:

- Realizar tareas "sencillas" y comunes: como redirigir a una página diferente a un usuario y renderizar una plantilla dada.

- Mostrar páginas de "listado" y "detalle" para un solo objeto. Por ejemplo una vista para presentar una lista de libros y una para presentar los detalles de un libro en especifico, la primera es una vista de listado, una página de objetos simples que muestra la lista de determinado modelo, mientras el segundo es un ejemplo de lo que llamamos vista "detallada".

- Presentar objetos basados en fechas en páginas de archivo de tipo día/mes/año, su detalle asociado, y las páginas "más recientes". Los archivos por día, mes, año del blog de Django http://www.djangoproject.com/weblog/ están construidos con ellas, como lo estarían los típicos archivos de un periódico.

- Permitir a los usuarios crear, actualizar y borrar objetos – con o sin autorización.

Agrupadas, estas vistas proveen interfaces fáciles y sencillas de usar para realizar las tareas más comunes que encuentran los desarrolladores.

Introducción a las clases genéricas

Las vistas genéricas basadas en clases, proveen una forma alternativa de implementar vistas como objetos Python en lugar de funciones. No remplazan a las funciones basadas en vista, pero poseen ciertas ventajas y diferencias si las comparamos con las vistas basadas en funciones:

- Organizan el código relacionado en métodos específicos HTTP (GET, POST, etc) para que puedan ser tratados por métodos específicos en lugar de tener que tratar cada uno por separado.
- Usan la técnica de orientación a objetos para crear "mixins" (herencia múltiple) que puede ser usada para factorizar el código en componentes comunes y reutilizables.

Como mencionamos en capítulos anteriores una vista es un llamable que toma una petición y retorna una respuesta. Pero una vista puede ser más que una función, Y Django provee ejemplos de algunas clases que pueden ser utilizadas como vistas. Estas permiten estructurar las vistas y rehusar el código aprovechando los mixins y la herencia. Existen vistas genéricas para realizar tareas simples, que veremos más adelante, sin embargo también sirven para diseñar estructuras personalizables y reutilizables que fácilmente se pueden adaptar a la mayoría de caso de uso.

Un poco de historia

La conexión y la historia de las vistas genéricas, vistas basadas en clase y las vistas genéricas basadas en clases-base, puede ser un poco confusa, sobre todo si es la primera vez que escuchas sobre ellas.

Inicialmente solo existían funciones basadas en vistas genéricas, Django pasaba la función en una petición HttpRequest y esperaba de vuelta una respuesta HttpResponse. Ese era todo el alcance que Django ofrecía.

El problema con las funciones genéricas basadas en vistas es que solo cubren los casos simples, pero no permiten extenderlas y personalizarlas mas allá de la simple configuración de opciones, limitando su utilidad en muchas aplicaciones del mundo real.

Las vistas genéricas basadas en clases, fueron creadas con el mismo objetivo que las basadas en funciones, hacer el desarrollo más sencillo. Por lo que la solución se implemento a través del uso de "mixins", que proveen un conjunto de herramientas, que dieron como resultado que las vistas genéricas se basaran en clases-base, para que fueran más extensibles y flexibles que su contraparte basadas en funciones.

Si usaste las funciones genéricas para crear vistas en el pasado y las encontraste limitadas y deficientes, *no* debes pensar que las vistas basadas en clases son su equivalente, ya que funcionan de modo diferente, piensa más en ellas, como un acercamiento fresco para solucionar el problema original, que la vistas genéricas tratan de solucionar, "hacer de el desarrollo aburrido, una tarea divertida".

El conjunto de herramientas que proveen las clases base y los "mixins" que Django usa para crear clases basadas en vistas genéricas, nos ayudan a realizar los trabajos comunes con una máxima flexibilidad, para situaciones simples y complejas.

Usando vistas basadas en clases

En su núcleo, una vista basada en una clase-base permite responder a diferentes métodos de petición HTTP, con diversos métodos de la instancia de una clase, en lugar de condicionalmente ramificar el código dentro de una simple función de vista.

Por lo que el código para manipular HTTP en una petición GET, en una función de vista sería algo como esto:

```
from django.http import HttpResponse

def mi_vista(request):

    if request.method == 'GET':
        # <la logica de la vista>
        return HttpResponse('resultado')
```

Mientras que en una vista basada en una clase-base, haríamos esto:

```
from django.http import HttpResponse
from django.views.generic import View

class  (View):

   def get(self, request):
       # <la logica de la vista>
       return HttpResponse('resultado')
```

Debido a que el resolvedor de URL de Django espera enviar la petición y los argumentos asociados a una función llamable no a una clase, la vistas basadas en clases provén un método interno llamado as_view(), que sirve como punto de entrada para enlazar la clase a la URL. El punto de entrada as_view() crea una instancia de la clase y llama al método dispatch(), (el despachador o resolvedor de URL) que busca la petición para determinar si es un GET, POST, etc, y releva la petición a un método que coincida con uno definido, o levante una excepción HttpResponseNotAllowed si no encuentra coincidencias.

Y así es como enlazamos la clase a la URL, usando el método as_view()

```
# urls.py
from django.conf.urls import url
from myapp.views import MiVista

urlpatterns = [
    url(r'^indice/', MiVista.as_view()),
]
```

Vale la pena observar que el método que devuelve es idéntico al que devuelve una vista basada en una función, a saber una cierta forma de HttpResponse. Esto significa que los atajos para los objetos shortcuts o TemplateResponse son válidos para usar dentro de una vista basada en clases.

También vale la pena mencionar que mientras que una vista mínima basada en clases, no requiere ningún atributo de clase para realizar su trabajo, los atributos de

una clase son útiles en muchos de los diseños de las clases-base. Hay dos maneras de configurar los atributos de una clase.

1. El primero está basado en la forma estándar de Python de sobrescribir atributos y métodos en las subclases. De modo que si una clase padre tiene un atributo saludo tal como este:

```
from django.http import HttpResponse
from django.views.generic import View

class VistaSaludo(View):
    saludo= "Buenos Días"

    def get(self, request):
        return HttpResponse(self.saludo)
```

   Puedes sobrescribirlo en una subclase así:

```
class VistaSaludoInformal(VistaSaludo):
    saludo= "Que onda"
```

2. La segunda opción es configurar los atributos de la clase como argumentos clave para el método as_view de django.views.generic.base.View.as_view, llamándolos en la URLconf así:

```
urlpatterns = [
    url(r'^acerca/', VistaSaludo.as_view(saludo="Que tal")),
]
```

■**Nota:** Mientras que una clase es instanciada en cada petición enviada a ella, los atributos de la clase fijados a través del punto de entrada del método *as_view ()* se configuran solamente una vez; cuando se importa la URLs.

Vista Base

Django proporciona varias vistas basadas en clases, las cuales se adaptan a una gran variedad de aplicaciones. Todas las vistas heredan de la clase-base View la cual maneja las conexiones de la vista y las URLs, a través del uso de métodos HTTP y otras características simples. Algunas de estas vistas son: RedirectView usada para simple redirecionamiento HTTP, TemplateView la cual extiende las clases base para poder renderizar una plantilla cualquiera.

Estas tres clases: **View, TemplateView** y **RedirectView** proveen muchas de las funcionalidades necesarias para crear vistas genéricas en Django. Puedes pensar en ellas como si fueran vista padre o superclases, las cuales pueden ser usadas en sí mismo o heredar de ellas.

Sin embargo no puede proveer todas las capacidades requeridas para un proyecto en general, en cuyo caso puedes usar los mixins y las vistas basadas en clases genéricas, como complemento.

Muchas de las vistas construidas sobre clases basadas en vistas, heredan de otras vistas genéricas también basadas en clases o de varios mixins. Debido a que esta cadena de herencia es muy importante, el manejo de ancestros de una clases se denomina (MRO). MRO por sus siglas en ingles para Method Resolution Orden, se encarga de resolver el orden que siguen los métodos en una clase.

View

View es la clase base maestra, las demás vistas heredan de esta clase base, que pertenece al paquete *class django.views.generic.base.View.*

Flujo de los métodos:

1. **dispatch():** El resolvedor de URL´s de la vista – es decir el método que valida el argumento de la petición, más los argumentos recibidos y devuelve la respuesta correcta HTTP.

   Por defecto es la implementación que inspecciona el método HTTP y tentativamente la delega al método que coincida con la petición HTTP; por ejemplo una petición GET será delegado a un método get(), un POST a un post(), y así sucesivamente.

2. **http_method_not_allowed():** Si la vista es llamada con un método HTTP no soportado, este método es llamado en su lugar.

   La implementación por defecto retorna un HttpResponseNotAllowed con una lista de métodos permitidos en texto plano.

3. **options():** Manejadores que responden a las peticiones OPTIONS HTTP. Retorna una lista de nombres permitidos al método HTTP para la vista

Ejemplo:

views.py:

```
from django.http import HttpResponse
from django.views.generic import View

class MiVista(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Hola, Mundo')
```

urls.py*:*

```
from django.conf.urls import url
from myapp.views import MiVista

urlpatterns = [
    url(r'^hola/$', MiVista.as_view(), name='mi-vista'),
]
```

Por defecto la lista de nombres de métodos HTTP que la vista `View puede aceptar son: 'get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace'.

TemplateView

La clase TemplateView renderiza una plantilla dada, con el contexto que contiene los parámetros capturados en la URL, esta clase pertenece al paquete class django.views.generic.base.TemplateView

Ancestros (MRO)

Esta vista hereda atributos y métodos de las siguientes vistas:

- django.views.generic.base.TemplateResponseMixin
- django.views.generic.base.ContextMixin
- django.views.generic.base.View

Flujo de los métodos:

1. **dispatch():** Valida la petición (ver arriba).
2. **http_method_not_allowed():** Verifica los métodos soportados.
3. **get_context_data():** Se encarga de pasarle el contexto (context) a la vista.

Ejemplo:

views.py:

```
from django.views.generic.base import TemplateView
from biblioteca.models import Libro

class PaginaInicio(TemplateView):
    template_name = "bienvenidos.html"
    def get_context_data(self, **kwargs):
    context = super(PaginaInicio, self).get_context_data(**kwargs)
    context['ultimos_libros'] = Libro.objects.all()[:5]
    return context
```

urls.py:

```
from django.conf.urls import url
from biblioteca.views import PaginaInicio

urlpatterns = [
    url(r'^$', PaginaInicio.as_view(), name='bienvenidos'),
]
```

La clase TemplateView rellena el **contexto** (a través de la clase django.views.generic.base.ContextMixin) con los argumentos clave capturados en el patrón URL, que sirve a la vista.

RedirectView

La clase RedirectView tal como su nombre lo indica, simplemente redirecciona una vista con la URL dada.

La URL dada puede contener un formato de estilo tipo diccionario, que será intercalado contra los parámetros capturados en la URL. Ya que el intercalado de palabras claves se hace *siempre* (incluso si no se le pasan argumentos), por lo que cualquier carácter como "%" (un marcador de posición en Python) en la URL debe ser escrito como %%" de modo que Python lo convierta en un simple signo de porcentaje en la salida.

Si la URL dada es None, Django retornara una respuesta HttpResponseGone (410).

Ancestros (MRO)

Esta vista hereda los métodos y los atributos de:

django.views.generic.base.View

Flujo de los métodos:

1. **dispatch()**
2. **http_method_not_allowed()**
3. **get_redirect_url():** Construye el URL del objetivo para el redireccionamiento. La implementación por defecto usa la url como la cadena de inicio para realizar la expansión mediante el marcador de posición % en la cadena usando el grupo de nombres capturados en la URL.

   Si no se configura el atributo url, mediante el método get_redirect_url() entonces Django intenta invertir el nombre del patrón, usando los argumentos capturados en la URL (usando los grupos con y sin nombre).

Si es una petición de un atributo query_string también se agregara a la cadena de consulta generada por la URL. Las subclases pueden ejecutar cualquier comportamiento que deseen, mientras que el método devuelva una cadena de redireccionamiento a una URL.

Los atributos de esta clase son:

- **url:** La URL para redireccionar la vista, en formato de cadena o un valor None para lanzar un error HTTP 410.

- **pattern_name:**El nombre de el patrón URL para redirecionar la vista. El redireccionamiento puede ser hecho usando los mismos args y kwargs que se pasan a las vistas.

- **permanent:** Se usa solo si el redireccionamiento debe ser permanente. La única diferencia aquí es el código de estado devuelto por la petición HTTP. Si es True, entonces el redireccionamiento utiliza el código de estado 301. Si es False, entonces el redireccionamiento utiliza el código de estado 302. Por defecto, permanent es True.

- **query_string:**Cualquier cosa que se le pase a la consulta usando el método GET a la nueva localización. Si es True, entonces la consulta se añade al final de la URL. Si es False, entonces la consulta se desecha. Por defecto, query_string es False.

Ejemplo:

views.py:

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView
from biblioteca.models import Libro

class ContadorLibrosRedirectView(RedirectView):
    permanent = False
    query_string = True
    pattern_name = 'detalle-libro'

    def get_redirect_url(self, *args, **kwargs):
        libro = get_object_or_404(Libro, pk=kwargs['pk'])
        libro.update_counter()
        return super(ContadorLibrosRedirectView,
            self).get_redirect_url(*args, **kwargs)
```

urls.py:

```
from django.conf.urls import url
from django.views.generic.base import RedirectView

from biblioteca.views import ContadorLibrosRedirectView, DetalleLibro

urlpatterns = [
    url(r'^contador/(?P<pk>[0-9]+)/$', ContadorLibrosRedirectView.as_view(),
        name='contador-libros'),
    url(r'^detalles/(?P<pk>[0-9]+)/$', DetalleLibro.as_view(),
        name='detalles-libro'),
    url(r'^ir-a-django/$', RedirectView.as_view(url='http://djangoproject.com'),
        name='ir-a-django'),
]
```

Vistas genéricas basadas en clases usando URLconfs

La manera más simple de utilizar las vistas genéricas es creándolas directamente en la URLconf. Si únicamente quieres cambiar algunos atributos en una vista basada en clases-base, puedes simplemente pasarle los atributos que quieres sobrescribir dentro del método as_view, ya que este es un llamable en sí mismo.

Por ejemplo, ésta es una URLconf simple que podrías usar para presentar una página estática “acerca de”, usando una vista genérica:

```
from django.conf.urls import url
from django.views.generic import TemplateView

urlpatterns = [
    url(r'^acerca/', TemplateView.as_view(template_name="acerca_de.html")),
]
```

Cualquier argumento pasado al método as_view sobrescribirá los atributos fijados en la clase. En este ejemplo, hemos configurado el nombre de la plantilla con la variable template_name en la URLconf, de la vista TemplateView. Un patrón similar se puede utilizar para sobrescribir atributos en la clase RedirectView.

Aunque esto podría parecer un poco "mágico" a primera vista, en realidad solo estamos usando la clase TemplateView, la cual renderiza una plantilla dada, con el contexto dado, sobrescribiendo el nombre de la plantilla y los atributos predefinidos en la clase base TemplateView.

Vistas genéricas basadas en clases usando subclases

La segunda forma más poderosa de usar las vistas genéricas es hacer que estas hereden de una vista sobrescribiendo sus atributos (tal como el nombre de la plantilla) o sus métodos (como get_context_data) en una subclase que proporcione nuevos valores o métodos. Considera por ejemplo una vista que muestre una plantilla acerca_de.html. Django posee una vista genérica que hace este trabajo, como lo vimos en el ejemplo anterior – TemplateView solo es necesario crear una subclase que sobrescriba el nombre de la plantilla así:

biblioteca/views.py

```
from django.views.generic import TemplateView

class VistaAcercaDe(TemplateView):
    template_name = "acerca_de.html"
```

Después lo único que necesitas es agregar la nueva vista a la URLConf. La clase TemplateView no es una función, así que apuntamos la URL usando un método interno as_view() de la clase en su lugar, el cual provee una entrada como si fuera una función a la vista basada en una clases-base.

biblioteca/urls.py

```
from django.conf.urls import url
from aplicacion.views import AboutView

urlpatterns = [
    url(r'^acerca/', VistaAcercaDe.as_view()),
]
```

Cualquier argumento pasado al método as_view() sobrescribira los definidos en la clase recién creada.

Vistas genéricas de objetos

La vista genérica TemplateView ciertamente es útil, pero las vistas genéricas de Django brillan realmente cuando se trata de presentar vistas del contenido de tu base de datos. Ya que es una tarea tan común, Django viene con un puñado de vistas genéricas incluidas que hacen la generación de vistas de listado y detalle de objetos increíblemente fácil.

Comenzaremos observado algunos ejemplos básicos, sobre como mostrar una lista de objetos usando la vista genérica basada en clases llamada ***ListView*** y como mostrar objetos de forma individual, usando la clase generica ***DetailView***.

Usaremos el modelo Editor creado en capítulos anteriores:

biblioteca/models.py

```
from django.db import models

class Editor(models.Model):
    nombre = models.CharField(max_length=30)
    domicilio = models.CharField(max_length=50)
    ciudad = models.CharField(max_length=60)
    estado = models.CharField(max_length=30)
    pais = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self): # __unicode__ en Python 2
        return self.nombre
```

Primero definimos una vista, para crear una lista de editores, usando una clase genérica llamada ListView:

biblioteca/views.py

```
from django.views.generic import ListView
from biblioteca.models import Editor

class ListaEditores(ListView):
    model = Editor
```

Como puedes ver la clase ListView pertenece a la clase django.views.generic.list.ListView la cual se encarga de presentar un listado de todos los objetos de un modelo, piensa en ListView como una consulta del tipo Editor.objets.all(). Cuando esta vista es ejecutada llama al método self.object_list el cual contiene una lista de objetos (usualmente, pero no necesariamente un queryset)

Después importamos la vista y la enlazamos directamente a la urls, usando el método as_view(), es como decirle a Django: esta clase es una vista:

urls.py

```
from django.conf.urls import url
from biblioteca.views import ListaEditores

urlpatterns = [
    url(r'^editores/$', ListaEditores.as_view(), name='lista-editores' ),
]
```

Ese es todo el código Python que necesitamos escribir, para presentar un listado de objetos de un modelo. Sin embargo, todavía necesitamos escribir una plantilla. Podríamos decirle explícitamente a la vista que plantilla debe usar incluyendo un atributo template_name, pero en la ausencia de una plantilla explícita Django inferirá una del nombre del objeto. En este caso, la plantilla inferida será "biblioteca/editor_list.html" – la parte “biblioteca” proviene del nombre de la aplicación que define el modelo, mientras que la parte “editor” es sólo la versión en minúsculas del nombre del modelo, mas el sufijo _list.

Nota: El cargador de plantillas esta activado de forma predeterminada, en el archivo de configuración, en la variable TEMPLATE_LOADERS, por lo que el directorio predeterminado donde Django buscara, las plantillas será: en el directorio: /ruta/a/proyecto/biblioteca/templates/biblioteca/editor_list.html.

Django por omisión busca un directorio con el nombre de la aplicación dentro del directorio de plantillas llamado templates, dentro de cada aplicación registrada.

Esta plantilla será renderizada con un contexto que contiene una variable llamada **object_list** la cual contiene todos los objetos "editor" del modelo.

Una plantilla muy simple podría verse de la siguiente manera:

biblioteca/templates/biblioteca/editor_list.html

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

{% block content %}
<h2>Editores</h2>

{% if object_list %}
   <ul>
   {% for editores  in object_list %}
      <li><a href="{% url 'detalles-editor' editores.pk %}">
          {{ editores.nombre }}</a></li>
   {% endfor %}
   </ul>
   {% else %}
      <p>No hay editores registrados.</p>
{% endif %}

{% endblock %}
```

(Observa que esta plantilla asume que existe una plantilla base (en el directorio superior templates), llamada "base.html", de la cual hereda, tal y como vimos en los ejemplos del *capítulo 4,* también asumen que existe una url llamada detalles-editor, la cual crearemos a continuación.)

Ciertamente obtener una lista de objetos con la clase genérica ListView es siempre muy útil, pero que pasa si queremos mostrar un solo objeto, por ejemplo los detalles de un determinado editor, en ese caso usamos la vista genérica DetailView, que se encarga de presentar los detalles de un objeto, ejecutando self.object el cual contendrá el objeto sobre el que la vista está operando.

Por ejemplo si quisiéramos mostrar un editor en particular, usaríamos la clase: DetailView, de esta manera:

biblioteca/views.py

```
from django.views.generic.detail import DetailView
from biblioteca.models import Editor

class DetallesEditor(DetailView):
   model = Editor
```

AL igual que con la vista anterior, solo necesitamos importarla y enlazar la vista a su respectiva URL así:

urls.py

```
from django.conf.urls import url
from biblioteca.views import DetallesEditor

urlpatterns = [
    url(r'^detalles/editor/(?P<pk>[0-9]+)/$', DetallesEditor.as_view(),
        name='detalles-editor' ),
]
```

Y por ultimo creamos la plantilla con el nombre por defecto que le asigna Django que es editor_detail.html:

biblioteca/templates/biblioteca/editor_detail.html

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

{% block content %}

<h2>Editor: {{ editor.nombre }}</h2>
<ul>
    <li>Domicilio: {{ editor.domicilio }}</li>
    <li>Ciudad: {{ editor.ciudad }}</li>
    <li>Estado: {{ editor.estado }}</li>
    <li>Pais: {{ editor.pais }}</li>
    <li>Sitio web: {{ editor.website }}</li>
</ul>
<p><a href="{% url 'lista-editores' %}">Lista de editores</a></p>

{% endblock %}
```

Observa que accedemos al contexto usando el nombre en minúsculas del modelo.

ListView y DetailView son las dos vistas basadas en clases genéricas que probablemente se usen más en el diseño de proyectos.

Eso es realmente todo en lo referente al tema. Todas las geniales características de las vistas genéricas provienen de cambiar los atributos fijados en la vista genérica. El Apéndice C documenta todas las vistas genéricas y todas sus opciones en detalle; el resto de este capítulo considerará algunas de las maneras más comunes en que puedes personalizar y extender las vistas genéricas basadas en clases.

Extender las vistas genéricas

No hay duda de que usar las vistas genéricas puede acelerar el desarrollo sustancialmente. En la mayoría de los proyectos, sin embargo, llega un momento en el que las vistas genéricas no son suficientes. De hecho, la pregunta más común que se hacen los nuevos desarrolladores de Django es cómo hacer para que las vistas genéricas manejen un rango más amplio de situaciones.

Afortunadamente, en casi cada uno de estos casos, hay maneras de simplemente extender las vistas genéricas para manejar un conjunto más amplio de casos de uso. Estas situaciones usualmente recaen en un puñado de patrones que se tratan en las secciones que siguen.

Crear contextos de plantilla “amistosos”

Tal vez hayas notado que el ejemplo de la plantilla editores almacena la lista de todos los editores en una variable llamada object_list. Aunque esto funciona bien, no es una forma “amistosa” para los autores de plantillas: ellos sólo tienen que “saber” que están trabajando con una lista de editores.

Bien, si estas tratando con un objeto de un modelo, el trabajo está hecho. Cuando estas tratando con un objeto o queryset, Django es capaz de rellenar el contexto usando el nombre de la clase en minúsculas de un modelo, mas **_list**. Esto es provisto además de la entrada predeterminada **object_list**, pero conteniendo exactamente los mismos datos, por ejemplo **editor_list**, es equivalente a **object_list**.

Si el nombre no es una buena idea, puedes manualmente cambiarlo en el contexto de la variable. El atributo **context_object_name** en una vista genérica específica el contexto de las variables a usar:

biblioteca/views.py

```
from django.views.generic import ListView
from biblioteca.models import Editor

class ListaEditores(ListView):
    model = Editor
    context_object_name = 'lista_editores'
```

Proporcionar útiles nombres de contexto (context_object_name) es siempre una buena idea, tus compañeros de trabajo que diseñan las plantillas te lo agradecerán.

Agregar un contexto extra

A menudo simplemente necesitas presentar alguna información extra aparte de la proporcionada por la vista genérica. Por ejemplo, piensa en mostrar una lista de todos los libros en cada una de las páginas de detalle de un editor.

La vista genérica DetailView, que pertenece a la clase django.views.generic.detail.DetailView provee el contexto a editores, ¿Pero cómo obtener información adicional en la plantilla?

La respuesta está en la misma clase DetailView, que provee su propia implementación del método get_context_data, la implementación por defecto simplemente agrega un objeto para mostrar en la plantilla, pero puede sobrescribirse aun más:

biblioteca/views.py

```
from django.views.generic import DetailView
from biblioteca.models import Editor, Libro

class DetallesEditor(DetailView):
    model = Editor
    context_object_name = 'editor'

    def get_context_data(self, **kwargs):
        # Llama primero a la implementación para traer un contexto
        context = super(DetallesEditor, self).get_context_data(**kwargs)
        # Agrega un QuerySet para obtener todos los libros
        context['lista_libros'] = Libro.objects.all()
        return context
```

Con esta vista obtenemos dos queryset, "editor" que muestra los detalles de un editor en específico y "lista_libros" que obtiene todos los libros de la base de datos.

■**Nota:** Por lo general *get_context_data* combina los datos del contexto de todas las clases padres con los de la clase actual. Para conservar este comportamiento en las clases donde se quiera alterar el comportamiento del contexto, asegúrate de llamar a *get_context_data* en la súper clase. Cuando ninguna de las dos clases trate de definir la misma clave, esto dará los resultados esperados. Sin embargo si cualquiera de las clases trata de sobrescribir la clave después de que la clase padre la ha fijado (después de llamar a súper) cualquiera de las clases hija necesitara explícitamente fijarla y asegurarse de sobrescribir todas las clases padres. Si tienes problemas, revisa el orden de resolución del método de una vista.

Vista para un subconjunto de objetos

Ahora echemos un vistazo más de cerca al argumento model que hemos venido usando hasta aquí. El argumento model especifica el modelo de la base de datos que usara la vista genérica, la mayoría de las vistas genéricas usan uno de estos argumentos para operar sobre un simple objeto o una colección de objetos. Sin embargo El argumento model no es la única forma de especificar los objetos que se mostraran en la vista, puedes especificar una lista de objetos usando como argumentos un queryset

```
from django.views.generic import DetailView
from biblioteca.models import Editor

class DetallesEditor(DetailView):
    context_object_name = 'editor'
    queryset = Editor.objects.all()
```

Especificando model = Editor es realmente un atajo para decir: queryset = Editor.objects.all(). Sin embargo, usando un queryset puedes filtrar una lista de objetos y puedes especificar los objetos que quieres que se muestren en la vista.

Para escoger un ejemplo simple, puede ser que quieras ordenar una lista de libros por fecha de publicación, con los libros más reciente al inicio:

```
from django.views.generic import ListView
from biblioteca.models import Libro

class LibrosRecientes(ListView):
    queryset = Libro.objects.order_by('-fecha_publicacion')
    context_object_name = 'libros_recientes'
```

Este es un ejemplo bastante simple, pero ilustra bien la idea. Por supuesto, tú usualmente querrás hacer más que sólo reordenar objetos. Si quieres presentar una lista de libros de un editor en particular, puedes usar la misma técnica:

```
from django.views.generic import ListView
from biblioteca.models import Libro

class LibroAcme(ListView):
    context_object_name = 'lista_libros_acme'
    queryset = Libro.objects.filter(editor__nombre='Editores Acme')
    template_name = 'biblioteca/lista_libros_acme.html'
```

Observa que además de filtrar un queryset, también estamos usando un nombre de plantilla personalizado. Si no lo hiciéramos, la vista genérica usaría la misma plantilla que la lista de objetos "genérica", que puede no ser lo que queremos.

También observa que ésta no es una forma muy elegante de hacer una lista de editores-específicos de libros. Si queremos agregar otra página de editores, necesitamos otro puñado de líneas en la URLconf, y más de unos cuantos editores no será razonable. Enfrentaremos este problema en la siguiente sección.

■**Nota:** Si obtienes un error 404 cuando solicitas /libros/acme/, para estar seguro, verifica que en realidad tienes un Editor con el nombre 'Editores Acme'. Las vistas genéricas proveen un parámetro extra allow_empty para estos casos. Mira el Apéndice D para mayores detalles.

Filtrado Dinámico

Otra necesidad muy común es filtrar los objetos que se muestran en una página de listado por alguna clave en la URLconf. Anteriormente codificamos el nombre de los editores en la URLconf, pero ¿qué pasa si queremos escribir una vista que muestre todos los libros por algún editor arbitrario?

Podemos "usar" la vista genérica ListView que posee un método get_queryset que pertenece a la clase django.views.generic.list.MultipleObjectMixin.get_queryset el cual sobrescribimos anteriormente, el cual retornaba el valor del atributo queryset, pero ahora le agregaremos más lógica.

La parte crucial para hacer este trabajo está en llamar a las vistas basadas en clases-base, ya que guardan algunas cosas útiles con self; tal como la petición (self.request) esta incluye la posición (self.args) el nombre base (self.kwargs) los argumentos capturados acorde a la URLconf.

Esta es una URLconf con un único grupo capturado:

urls.py

```
from django.conf.urls import url
from biblioteca.views import ListaLibrosEditores

urlpatterns = [
    url(r'^libros/([\w-]+)/$', ListaLibrosEditores.as_view(), name='lista-libros-editor' ),
]
```

A continuación, escribimos la vista ListaLibrosEditores anterior:

biblioteca/views.py

```
from django.shortcuts import get_object_or_404
from django.views.generic import ListView

from biblioteca.models import Libro, Editor

class ListaLibrosEditores(ListView):
    template_name = 'biblioteca/lista_libros_por_editores.html'

    def get_queryset(self):
        self.editor = get_object_or_404(Editor, nombre=self.args[0])
        return Libro.objects.filter(editor=self.editor)
```

Como puedes ver, es sencillo agregar más lógica a la selección del queryset, en este caso filtrándolo; si quieres puedes usar self.request.user para filtrar usando el usuario actual o realizar otra lógica más compleja.

También puedes agregar un editor dentro del contexto, así puedes utilizarlo en la plantilla al mismo tiempo:

```
    # ...

    def get_context_data(self, **kwargs):
        # Llama primero a la implementación para traer el contexto
        context = super(ListaLibrosEditores, self).get_context_data(**kwargs)
        # Se agrega el editor
        context['editor'] =self.editor
        return context
```

Para llamar a esta vista en la plantilla "editor_detail.html" usa la siguiente línea:

```
<p><a href="{% url 'lista-libros-editor' editor.nombre %}">Libros publicados</a></p>
```

Realizar trabajo extra

El último patrón común que veremos involucra realizar algún trabajo extra antes o después de llamar a la vista genérica.

Imagina que tenemos un campo ultimo_acceso en nuestro modelo Autor que usamos para tener un registro de la última vez que alguien vio ese autor.

biblioteca/models.py

```
from django.db import models

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')
    ultimo_acceso  = models.DateTimeField(blank=True, null= True)
```

La vista genérica basada en la clase DetailView, por supuesto, no sabría nada sobre este campo, pero una vez más, fácilmente podríamos escribir una vista personalizada para mantener ese campo actualizado.

Primero, necesitamos agregar una pequeña parte de detalle sobre el autor en la URLconf para que apunte a la vista personalizada:

urls.py

```
from django.conf.urls import url
from biblioteca.views import VistaDetallesAutor

urlpatterns = [
    #...
    url(r'^autores/(?P<pk>[0-9]+)/$', VistaDetallesAutor.as_view(),
        name='detalles-autor'),
]
```

Nota: La URLconf aquí usa un nombre de grupo pk – este nombre, es el nombre predeterminado que DetailView usa para encontrar el valor de una clave primaria que se usa para filtrar el queryset (que no es más que la clave primaria o primary key.)

Si quieres llamar esta vista con otro nombre de grupo, puedes fijarlo a pk_url_kwarg en la vista.

Después escribimos la vista – get_object es un método que recupera un objeto, simplemente sobreescribe y envuelve la llamada.

biblioteca/views.py

```
from django.views.generic import DetailView
from django.utils import timezone

from biblioteca.models import Autor

class VistaDetallesAutor(DetailView):
    queryset = Autor.objects.all()

    def get_object(self):
        # LLama a la superclase
        objeto = super(VistaDetallesAutor, self).get_object()
        # Graba la fecha de el último acceso
        objeto.ultimo_acceso = timezone.now()
        objeto.save()
        # Retorna el objeto
        return objeto
```

Introducción a los mixins

Los mixins son una forma de herencia múltiple, donde los comportamientos y los atributos de múltiples clases padre, pueden heredarse y combinarse en una única clase hija.

Por ejemplo en las vistas genéricas basadas en clases existe un mixin llamado *TemplateResponseMixin* cuyo propósito central es definir el método render_to_response(). Cuando se combina con el comportamiento de la clase base View, el resultado es una clase *TemplateView* que enviara peticiones a los métodos que coincidan con la petición del patrón (un comportamiento definido en la clase base View) en el método render_to_response() y que utiliza un atributo como el

nombre de una plantilla para retornar un objeto mediante TemplateResponse (un comportamiento definido en el mixin TemplateResponseMixin.)

Los mixins son una excelente manera de reutilizar el código a través de múltiples clases, pero vienen con un cierto costo. Cuanto más los utilizas mas se dispersa el código, lo que dificulta leer lo que hace exactamente una clase hija y complica aún más saber qué métodos remplazan los mixins si es que estas usando la herencia en subclases con una cierta profundidad.

Observa también que puedes heredar solamente de una vista genérica - es decir, sólo una clase padre puede heredar de una vista y el resto (eventualmente) deben ser mixins. Si intentas heredar de más de una clase que herede de View – por ejemplo, tratando de usar un formulario en la cima de una lista y combinándola con ProcessFormView y ListView – no trabajará según lo esperado.

Usando un mixin en vistas genéricas

Veamos ahora como usar un simple mixin llamado SingleObjectMixin que se encarga de recuperar un solo objeto, con una vista genérica ListView que como vimos anteriormente presenta una lista de objetos de un determinado modelo.

La vista genérica ListView ofrece paginación incorporada, para la lista de objetos de un modelo, usando el atributo paginate_by, pero a lo mejor lo que quieres paginar es una lista de objetos que están enlazados (por una clave foránea por ejemplo) a otro objeto. En el modelo Editor que vimos anteriormente, para paginar una lista de libros por un editor en específico, podríamos hacerlo de la siguiente forma.

Combinando una vista ListView con un mixin SingleObjectMixin, a fin de que el queryset para la lista paginada de libros cuelgue de un simple objeto editor. Para hacer esto necesitamos primero obtener dos querysets diferentes:

- **Libro**: queryset para usar en ListView.

  Puesto que tenemos acceso a la lista de libros de un editor que queremos listar, podemos simplemente sobrescribir el método get_queryset () y utilizar el manejador para usar los editores del campo foráneo Libro en relación inversa.

- **Editores:** un queryset para usar con get_object().

  Confiaremos en la implementación predeterminada del método get_object() para traer el objeto correcto Editor. Sin embargo, necesitamos explícitamente pasarle un argumento al queryset porque de otra manera la implementación predeterminada de get_object() llamara al método get_queryset() el cual sobrescribirá los objetos Libro devueltos en lugar de el Editor.

Con esto en mente, ahora podemos escribir la vista:

biblioteca/views.py

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin

from biblioteca.models import Editor

class DetalleEditores(SingleObjectMixin, ListView):
    paginate_by = 3
    template_name = "biblioteca/detalles_editores.html"
```

```
    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Editor.objects.all())
        return super(DetalleEditores, self).get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super(DetalleEditores, self).get_context_data(**kwargs)
        context['editor'] = self.object
        return context

    def get_queryset(self):
        return self.object.libro_set.all()
```

Fíjate cómo colocamos self.object dentro del método get() para usarlo más adelante dentro del método get_context_data() y obtener un método get_queryset(). Si no usamos el atributo template_name para configurar el nombre de la plantilla, Django usara el valor por defecto para ListView la cual en este caso es "biblioteca/libro_list.html" porque es una lista de libros; ListView no sabe nada acerca de el mixin SingleObjectMixin, así que no tiene ninguna pista sobre que esta vista es una lista de libros de acuerdo a un editor predeterminado.

Observa que el atributo paginate_by es deliberadamente pequeño en este ejemplo, para que no tengas que crear un buen lote de libros para ver en funcionamiento la paginación.

Esta es la plantilla que usa:

biblioteca/templates/biblioteca/ detalles_editores.html

```
{% extends "base.html" %}
{% block content %}
    <h2>Editor {{ editor.nombre }}</h2>

    <ol>
        {% for libro in page_obj %}
            <li>{{ libro.titulo }}</li>
        {% endfor %}
    </ol>

    <div class="pagination">
        <span class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number }}">anterior</a>
            {% endif %}
            <span class="current">
                Pagina {{ page_obj.number }} de {{ paginator.num_pages }}.
            </span>
            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">siguiente</a>
            {% endif %}
        </span>
    </div>
{% endblock %}
```

Esta es la url que puedes usar para llamar la vista:

```
urls.py
from django.conf.urls import url
from biblioteca.views import DetalleEditores

urlpatterns = [
    #...
   url(r'^detalle/editores/(?P<pk>[0-9]+)/$', DetalleEditores.as_view(),
      name='detalle-editores' ),
]
```

Y se puede llamar con la clave primaria de un editor, en una plantilla de detalles :

```
<p><a href="{% url 'detalle-editores' editor.pk %}">Editores</a></p>
```

El uso de mixins y vistas genéricas es una buena forma de extender las vistas basadas en clases, en el ejemplo anterior observamos en acción un simple mixins llamado SingleObjectMixin que se encarga de traer un objeto, sin embargo Django cuenta con una conveniente cantidad de mixins repartidos en las siguientes categorías, que se explican a sí mismos:

- Simple mixins
- Single object mixins
- Multiple object mixins
- Editing mixins
- Date-based mixins

Envolviendo el método as_view() con mixins

Una forma de aplicar un comportamiento común a muchas clases es escribir un mixin que envuelva el método *as_view ().*

Por ejemplo, si tienes muchas vistas genéricas que necesites decorar con un método *login_required* () lo podrías implementar usando un mixin como este:

```
from django.contrib.auth.decorators import login_required

class RequiereLogin(object):

   @classmethod
   def as_view(cls, **initkwargs):
      vista = super(RequiereLogin, cls).as_view(**initkwargs)
      return login_required(vista)

class MiVista(RequiereLogin,  ...):
   # Esta es la vista genérica
   ...
```

Manejando formularios con vistas basadas en clases genéricas

Una vista basada en una función que maneja un formulario, luce así:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import MyForm

def mivista(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <proceso el formulario con cleaned data>
            return HttpResponseRedirect('/success/')
    else:
            form = MyForm(initial={'key': 'value'})

    return render(request, 'formulario.html', {'form': form})
```

De igual forma una vista basada en una clase base, se ve así:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views.generic import View

from .forms import MyForm

class MiFormulario(View):
    form_class = MyForm
    initial = {'key': 'value'}
    template_name = 'formulario.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <proceso el formulario con cleaned data>
            return HttpResponseRedirect('/success/')

        return render(request, self.template_name, {'form': form})
```

Como puedes observar, este es un caso muy simple del uso de clases genéricas para el manejo de formularios, pero te darás cuenta enseguida de las ventajas de usar este enfoque basado en clases, ya que tendrías la opción de modificar esta vista para requisitos particulares, personalizando y sobrescribiendo los atributos de la vista, por ejemplo form_class, template_name a través de la configuración de la URLconf, o de una subclase y también podrías reemplazar uno o más métodos (¡o todos!).

Ejemplo de un formulario usando una clase genérica

Como se menciona anteriormente las vistas genéricas de Django brillan realmente cuando se necesitan presentar datos, sin embargo también brillan cuando es necesario guardar y procesar datos mediante formularios Web.

Al trabajar con modelos podemos crear automáticamente formularios a partir de un modelo, usando vistas genéricas basadas en clases (en el capítulo 8 vimos un ejemplo).

Esta es la forma en que las puedes utilizar:

- Si se da el atributo de un modelo, ese modelo de clase será utilizada.
- Si get_object () devuelve un objeto, la clase de ese objeto será utilizada.
- Si se da un queryset, el modelo para ese queryset será utilizado.

Las vistas para los modelos de un formulario proveen un método form_valid que sobrescribe el modelo automáticamente. Puedes reemplazar esto si necesitas algún requisito en especial.

No necesitas proveer un método success_url para una vista tipo CreateView o UpdateView ya que usan el método get_absolute_url() de el modelo, si este está disponible.

Si quieres usar un formulario personalizado con la clase ModelForm (como una instancia para agregar validación) simplemente fija el valor form_class en la vista.

■**Nota:** Cuando especifiques una clase de un formulario personalizada, es necesario especificar el modelo usando una clase de ModelForm.

Para ver las clase genéricas en acción, lo primero que vamos hacer es agregar un método get_absolute_url() a la clase Autor del modelo, para así usarlo como redirecionamiento por defecto:

biblioteca/models.py

```
from django.db import models
from django.core.urlresolvers import reverse

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    # Omitimos los demas campos y métodos.

    def get_absolute_url(self):
        return reverse('detalles-autor', kwargs={'pk': self.pk})
```

Ahora podemos llamar a la clase CreateView y a sus amigos para que hagan el trabajo duro.

Observa que lo único que necesitamos es configurar las vistas genéricas basadas en clases-base aquí; no tenemos que escribir ninguna lógica nosotros mismos, sin embargo por convención agregamos los formularios en una vista especial llamada por convención **forms.py,** en el mismo nivel que models.py :

biblioteca/forms.py

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy

from biblioteca.models import Autor

class CrearAutor(CreateView):
    model = Autor
    fields = ['nombre', 'apellidos', 'email',]

class ActualizarAutor(UpdateView):
    model = Autor
    fields = ['nombre', 'apellidos', 'email',]

class BorrarAutor(DeleteView):
    model = Autor
    success_url = reverse_lazy('lista-autor')
```

Nota: Observa que usamos el método reverse_lazy() en la última clase, el cual es útil para cuando se necesita utilizar una url inversa, antes de que se cargue la URLConf de el proyecto.
El atributo fields trabaja de la misma forma que un atributo fields en una clase interna Meta dentro de una clase ModelForm. A menos que definas un formulario de otra forma el atributo es requerido y la vista lanzara una excepción del tipo ImproperlyConfigured si no lo encuentra.

Finalmente enlazamos las nuevas vistas basadas en clases para Crear, Actualizar y Borrar objetos, (CRUD por sus siglas en ingles: Create, Update y Delete) en la URLconf:

urls.py

```
from django.conf.urls import url
from biblioteca.forms import CrearAutor, ActualizarAutor, BorrarAutor

urlpatterns = [
    # ...
    url(r'autor/agregar/$', CrearAutor.as_view(), name='agregar-autor'),
    url(r'autor/(?P<pk>[0-9]+)/$', ActualizarAutor.as_view(), name='actualizar-autor'),
    url(r'autor/(?P<pk>[0-9]+)/borrar/$', BorrarAutor.as_view(), name='borrar-autor'),
]
```

Esta vistas heredan del mixin SingleObjectTemplateResponseMixin el cual usa el método template_name_suffix para construir el nombre de la plantilla con el atributo template_name basado en el nombre del modelo.

En este ejemplo:

- **CreateView** y **UpdateView** usan la misma plantilla: "biblioteca/autor_form.html"
- **DeleteView** usa la plantilla "biblioteca/autor_confirm_delete.html"

Si quieres especificar nombres diferentes para cada plantilla de la clase CreateView y UpdateView, puedes configurarlos mediante el atributo "template_name" como en cualquier vista basada en clases.

biblioteca/autor_form.html

```
<html>
<head>
   <title>Agregar autor</title>
</head>
<body>
   <h1>Agregar autor</h1>

   {% if form.errors %}
      <p style="color: red;">
         Por favor corrige lo siguiente:
      </p>
   {% endif %}

   <form action="" method="post">{% csrf_token %}
      <div class="field">
         {{ form.nombre.errors }}
         <label for="id_nombre">Nombre:</label>
         {{ form.nombre }}
      </div>
      <div class="field">
         {{ form.apellidos.errors }}
         <label for="id_apellidos">Apellidos:</label>
         {{ form.apellidos }}
      </div>
      <div class="field">
         {{ form.email.errors }}
         <label for="id_email">E-mail:</label>
         {{ form.email }}
      </div>
      <input type="submit" value="Enviar">
   </form>

</body>
</html>
```

Como practica, crea la plantilla faltante.

Si sigues los ejemplos interactivamente, en este punto de puedes preguntar ¿Cómo protejo mis vistas para que solo usuarios autentificados puedan acceder a ellas, ya que sería bastante "descuidado" que cualquiera pudiera crear borrar o actualizar objetos de la base de datos, sin autorización ? La respuesta es usando decoradores, ya sea en la vista misma o en la URLconf.

Django te provee de algunos atajos dedicados especialmente a cubrir estas necesidades, puedes usar: login_required o permission_required. Porque estos temas son una necesidad muy común veamos su uso.

Decorando vistas de una clase-base

La extensión de vistas basadas en clases no se limita a usar solamente mixins. También puedes utilizar decoradores. Puesto que las vistas basadas en clases no son funciones, necesitas decorarlas de forma diferente dependiendo de si estás utilizando el método as_view o está creando una subclase de una clase.

Decorando vistas de una clase-base

La forma más simple de decorar una vista basada en una clase, es decorar el resultado de el método *as_view().* El lugar más sencillo para hacer esto es en la URLconf donde se despliega la vista:

```
from django.contrib.auth.decorators import login_required
from django.views.generic import TemplateView

from biblioteca.forms import CrearAutor

urlpatterns = [
    #
    url(r'agregar/autor/$', permission_required(CrearAutor.as_view()),
        name='agregar-autor'),
  ]
```

Esta aproximación aplica únicamente a decoradores por-instancia. Si quieres que cada instancia de una vista sea decorada, necesitas usar un acercamiento diferente

Decorando una clase

Para decorar cada instancia de una vista basada en clases, necesitas decorar la definición de la clase misma. Para hacer esto aplica el decorador a el método dispatch() de la clase.

Un método sobre una clase no equivale realmente a una función independiente, así que solo puedes aplicar un decorador a un método de una función –- por lo que necesitas transformarlo en un decorador primero. El decorador @method_decorator transforma un decorador de una función en un decorador de un método a fin de que puede ser usado sobre una instancia de un método. Por ejemplo:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator

from django.views.generic import TemplateView

class Vista Protegida(TemplateView):
    template_name = 'pagina-secreta.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(Vista Protegida, self).dispatch(*args, **kwargs)
```

En este ejemplo, cada instancia de Vista Protegida, tendrá protección de login.

■**Nota:** El *method_decorator* pasa ***args** y ****kwargs*** como parámetros al método del decorador de la clase. Si el método no valida el conjunto de parámetros compatibles levantará una excepción del tipo TypeError.

Soporte para Apis

Supongamos que alguien quiere acceder a nuestra biblioteca de libros sobre HTTP, usando la vista como una API. La API del cliente se conectaría de vez en cuando y descarga la lista de libros publicados desde su última visita. Pero si no se ha publicado ningún libro desde la última vez, sería una pérdida de CPU y de ancho de banda obtener todos los libros de la base de datos, renderizar una respuesta completa y enviársela al cliente. No sería preferible preguntarle a la API cuales son los libros recientemente publicados.

Para hacer este trabajo, mapeamos la URL a la vista que obtendrá la lista de libros:

```
from django.conf.urls import url
from biblioteca.views import VistaLibrosRecientes

urlpatterns = [
    url(r'^libros/$', VistaLibrosRecientes.as_view(), name = 'ultimos-libros'),
]
```

Y creamos la clase para la vista:

```
from django.http import HttpResponse
from django.views.generic import ListView

from biblioteca.models import Libro

class VistaLibrosRecientes(ListView):
    model = Libro
    template_name = 'ultimos_libros.html'

    def head(self, *args, **kwargs):
        ultimos_libros = self.get_queryset().latest('fecha_publicacion')
        response = HttpResponse('')
        # Formato de datos RFC 1123
        response['modificados'] = ultimos_libros.fecha_publicacion.strftime(
            '%a, %d %b %Y %H:%M:%S GMT')
    return response
```

Si la vista es accesada por una petición GET una simple lista de objetos será devuelta como respuesta (usando la plantilla “ultimos_libros.html”) Pero si el cliente nos envía una petición HEAD, la respuesta tendrá un cuerpo vacio y la cabecera de la última modificación indicara los libros que se publicaron recientemente. Basados en esta información, el cliente puede o no descargar la lista completa de objetos.

¿Qué sigue?

En este capítulo hemos examinado sólo un par de las vistas genéricas que incluye Django, pero las ideas generales presentadas aquí deberían aplicarse a cualquier vista genérica basada en clases. El Apéndice C cubre todas las vistas disponibles en detalle, y es de lectura obligada si quieres sacar el máximo provecho a esta poderosa característica de Django.

Aquí concluye la sección del libro dedicada al "uso avanzado de Django". En el próximo capítulo cubriremos el despliegue de aplicaciones en Django.

CAPÍTULO 12

Desplegando Django

Este capítulo cubre el último paso esencial para construir una aplicación Django: eldespliegue en un servidor de producción.

Si has estado siguiendo los ejemplos presentados, probablemente has estado usando *runserver,* como ya sabes este comando inicia el servidor web y hace que las cosas sean realmente muy fáciles – con runserver, no tienes que preocuparte por instalar un servidor Web.

Sin embargo *runserver* está diseñado únicamente para ser usado en la etapa de desarrollo de forma local, no para exponerlo en un sitio público Web. Para desplegar una aplicación Django, es necesario utilizar un servidor Web poderoso tal como *Apache*. En este capítulo te mostraremos como hacerlo – pero primero, es necesario comprobar una lista de cosas que hacer en el código antes de llevarlo a un entorno de producción.

Prepara tu código base para producción

Afortunadamente el comando runserver es bastante parecido a un servidor "real", por lo que no necesitas realizar muchos cambios a tu aplicación Django para dejarla lista para producción. Sin embargo hay algunas *cosas esenciales* que necesitas conocer antes de servirla en producción.

Desactiva el Modo Debug

Cuando creamos un proyecto en el *capítulo 2*, el comando django-admin.py startproject creó un archivo llamado settings.py el cual contiene la variable DEBUG fijada en True por defecto, es decir que esta en modo de depuración. Muchas de las partes internas de Django comprueban esta configuración y cambian su comportamiento si el modo DEBUG esta activado. Por ejemplo, si DEBUG está fijado en True, entonces:

Todas las consultas a la base de datos se guardan en memoria como objetos django.db.connection.queries. Como puedes imaginar, ¡esto consume mucha memoria!

Cualquier error 404, renderizara una página especial, a decir una página no encontrada (cubierta en el capítulo 3) más que retornar una apropiada respuesta 404. Esta página contiene información potencialmente sensible y *no* debe ser expuesta al público en Internet.

Cualquier excepción no atrapada en tu aplicación Django – desde errores básicos en la sintaxis Python, errores de la base de datos o en la sintaxis de la plantilla – serán renderizados por las páginas de errores bonitas de Django que probablemente adoras. Esta página contiene *más* información sensible incluso que las páginas 404, por lo que *nunca* deben ser expuestas al público.

Resumiendo, la variable de configuración DEBUG = True le dice a Django que asuma que únicamente desarrolladores confiables están usando el sitio. La internet está llena de personas poco fiables, y la primera cosa que debes hacer cuando estés

preparando tu aplicación para el despliegue en producción es fijar la variable DEBUG a False.

Desactiva el Modo Debug en las plantillas

De igual forma, es necesario fijar la variable TEMPLATE_DEBUG a False en producción. Esto para desactivar el modo depuración de las plantillas. Ya que si está fijada en True, esta configuración le dice al sistema de plantillas de Django que guarde información extra acerca de cada plantilla, para mostrarla en las útiles páginas de error, si el modo DEBUG = True (esta activado).

Implementa una plantilla 404

Si DEBUG es True, Django mostrara una muy útil página de error 404. Pero si DEBUG es False, hará algo completamente diferente: renderizara una plantilla llamada 404.html en la raíz del directorio de plantillas. Entonces, cuando estás listo para el despliegue, necesitas crear esta plantilla y ponerle algo útil, como un mensaje de "Página no encontrada".

Aquí está un ejemplo de una página 404.html, puedes usarla como el punto de partida, para crear tu propia plantilla. La plantilla asume que estas usando la herencia de plantillas y tiene definida una plantilla base.html con bloques llamados titulo y contenido.

```
404.html
{% extends "base.html" %}
{% block title %}Página no encontrada {% endblock %}

{% block content %}
    <h1>Página no encontrada</h1>
   <p>Lo sentimos, pero la página que buscas no ha sido encontrada.</p>
{% endblock %}
```

Para probar que la plantilla 404.html funciona, solo cambia el modo DEBUG a False y visita una URL que no exista. (Esto trabaja de igual forma en el servidor de desarrollo con runserver, que en un servidor de producción.)

Implementa una plantilla 500

De igual forma si DEBUG es False, Django no mostrara la útil página de traza de errores, en caso de excepciones no manejadas. En su lugar mostrara y renderizara una plantilla llamada 500.html. Tal como la página 404.html, esta plantilla debe de localizarse en la raíz del directorio de plantillas.

Hay una ligera trampa acerca de las páginas 500.html. Nunca puedes estar seguro por qué se renderiza esta plantilla, así que no deberías hacer nada que requiere una conexión a la base de datos o confiar en cualquier parte potencialmente rota de tu infraestructura. (Por ejemplo, no deberías usar etiquetas personalizadas en la plantillas 500.) Si usas la herencia de plantillas, entonces la plantilla padre, no debería poder conectarse a la infraestructura potencialmente rota. Por consiguiente, la mejor forma de aprovechar esto, es evitar la herencia de plantillas y usar algo muy simple y solo en HTML. Aquí hay un ejemplo de una página 500.html, como un punto de partida, para que diseñes la tuya:

```
500.html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
   <title>Página no disponible</title>
</head>

<body>
    <h1>Página no disponible</h1>
   <p>Lo sentimos, pero la página pedida no está disponible por que
      el servidor tiene hipo.</p>

    <p>Los desarrolladores han sido notificados, así que vuelva a revisar más
        tarde</p>
</body>
</html>
```

Establece errores de alerta

Cuando tu sitio creado con Django está corriendo y se lanza una excepción, necesitas estar al tanto, para poder corregir cualquier defecto. Por omisión, Django está configurado para enviar un e-mail a los desarrolladores del sitio, cuando el código lanza una excepción; pero necesitas hacer algunas cosas para configurarlo.

- Primero cambia la configuración ADMINS para incluir la dirección de e-mail, con las direcciones de cada una de las personas que necesitan ser notificadas. Esta configuración es una tupla del tipo (nombre, email), la tupla puede ser así:

```
ADMINS = (
('John Lennon', 'jlennon@example.com'),
('Paul McCartney', 'pmacca@example.com'),
)
```

- Segundo, asegúrate de que tu servidor este configurado para enviar e-mails. Configurar postfix, sendmail o cualquier otro servidor de e-mails, esta fuera del alcance de este libro, pero del lado de Django, debes de asegurarte de configurar EMAIL_HOST y establecer el "hostname" apropiado para tu servidor de correo. Si lo estableces a 'localhost' por defecto, trabajara fuera de la caja en muchos entornos de servicios compartidos. También es necesario que establezcas los parámetros de las siguientes configuraciones: EMAIL_HOST_USER, EMAIL_HOST_PASSWORD, EMAIL_PORT o EMAIL_USE_TLS, dependiendo de la complejidad de tu infraestructura.

- Por último, también establece el prefijo con EMAIL_SUBJECT_PREFIX el cual controla el nombre que Django usa delante de los errores en los correos electrónicos. Por defecto está establecido en '[Django]'.

Establece alertas para enlaces rotos

Si tienes la clase CommonMiddleware instalada (en tu archivo de configuración en la variable MIDDLEWARE_CLASSES que incluye 'django.middleware.common.CommonMiddleware', lo cual ocurre por defecto) tienes la opción de recibir un e-mail cada vez que alguien visita una página creada con Django que lanze un error 404 con una referencia a no-vacía – esto es cada vez que encuentre enlaces rotos. Si quieres activar esta característica, establece SEND_BROKEN_LINK_EMAILS a True (esta es False por defecto) y establece en la configuración MANAGERS a la persona o personas que recibirán esos e-mails de enlaces rotos. MANAGERS usa la misma sintaxis que ADMINS, un tupla. Por ejemplo:

```
MANAGERS = (
    ('George Harrison', 'gharrison@example.com'),
    ('Ringo Starr', 'ringo@example.com'),
)
```

Observa que los e-mails de errores pueden llegar a ser muy molestos; no son para todo el mundo.

Como usar diferentes configuraciones para producción

Hasta ahora en este libro, hemos tratado únicamente con un simple archivo de configuraciones s*ettings.py* generado por el comando django-admin.py startproject. Pero como estamos listos para desplegarlo, probablemente nos encontremos con la necesidad de usar múltiples archivos de configuraciones para mantener el entorno de desarrollo aislado del ambiente de producción.

Por ejemplo, probablemente no quieras cambiar DEBUG de False a True cada vez que quieras probar los cambios al código, en tu maquina de forma local. Django hace esto muy sencillo, permitiéndote usar múltiples archivos de configuraciones.

Si quieres organizar tus archivos de configuraciones, dividiéndolos en "desarrollo" y "producción", puedes lograrlo usando una de las siguientes tres formas.

- Establece dos archivos de configuraciones completos, de forma independiente.
- Establece un archivo de configuraciones "base" (uno para desarrollo) y un segundo (para producción) el archivo de configuraciones simplemente importa del primero y define lo que necesita y lo que debe sobrescribirse.
- Usa únicamente un archivo de configuraciones y deja que Python se encargue de la lógica y haga los cambios a las configuraciones, basado en el contexto.

Veamos estas tres opciones, una por una.

Primero, la forma más básica para aprovechar esto, es definir dos archivos separados. Si estas siguiendo esto, ya tienes un archivo settings.py. Ahora solo es necesario realizar una copia llamada settings_production.py.(Si no te gusta el nombre, puedes llamarlo como quieras) En este nuevo archivo, cambia las variables necesarias como DEBUG, etc.

La segunda forma de aprovechar esto es muy parecida, pero reduce algo de redundancia, En lugar de tener dos archivos de configuraciones con contenido similar, tratamos solo con un archivo "base" y creamos otro archivo que lo importe.

Por ejemplo:

```
# settings.py
DEBUG = True
TEMPLATE_DEBUG = DEBUG
DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'devdb'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_PORT = ''
# ...

# settings_produccion.py
from settings import *

DEBUG = TEMPLATE_DEBUG = False
DATABASE_NAME = 'production'
DATABASE_USER = 'app'
DATABASE_PASSWORD = 'dejameentrar'
```

Aquí tenemos que settings_production.py importa cualquier cosa de settings.py y solo redefine las configuraciones que son usadas especialmente en producción. En este caso DEBUG está fijado en False, pero también hemos fijado diversos parámetros, como el acceso a la base de datos para configurarla en producción. (Mas adelante te mostraremos como redefinir *cualquier* configuración, no únicamente las básicas como DEBUG.)

Finalmente, la forma más concisa para lograr tener dos entornos de configuraciones es usar un solo archivo, que se ramifique basado en el entorno. Una de las formas de lograr esto es comprobar el actual "hostname." Por ejemplo:

```
# settings.py
import socket

if socket.gethostname() == 'my-laptop':
   DEBUG = TEMPLATE_DEBUG = True
else:
   DEBUG = TEMPLATE_DEBUG = False
 # ...
```

Aquí, importamos el modulo socket de la librería estándar de Python y lo usamos para comprobar el nombre actual del sistema. Podemos comprobar el "nombre" para determinar si el código está siendo ejecutado en un servidor de producción.

La lección central, es que el archivo de configuraciones es *solo código Python*. Que puede importar otros archivos, puede ejecutar lógica arbitraria, etc. Solo asegúrate de que el código Python en tu archivo de configuraciones sea a prueba de balas. Si lanza una excepción, Django probablemente se estrellara de fea manera.

RENOMBRAR settings.py

Siéntete libre de renombrar tu archivo settings.py a settings_dev.py o settings/dev.py o foobar.py – A Django no le importa como lo llames, solo necesita saber que archivo estas usando para usarlo como configuración.

Solo ten en cuenta que si renombras el archivo settings.py que es generado por el comando django-admin.py startproject, te encontraras con que manage.py lanza un

mensaje de error, diciendo que no puede encontrar el archivo de configuraciones. Esto es debido a que trata de importar un modulo llamado settings. Puedes arreglar esto editando manage.py y cambiándole el nombre de settings por el nombre de tu modulo o usando django-admin.py en lugar de manage.py. En este último caso, necesitas fijar DJANGO_SETTINGS_MODULE con las variables de entorno de la ruta de búsqueda Python en tu archivo de configuraciones (por ejemplo 'misitio.settings')

Cuando usas Django tienes que indicarle qué configuración estás usando. Haz esto mediante el uso de de la variable de entorno DJANGO_SETTINGS_MODULE. El valor de DJANGO_SETTINGS_MODULE debe respetar la sintaxis de rutas de Python (por ej. misitio.settings. Observa que el módulo de configuración debe de encontrarse en la ruta de búsqueda para las importaciones de Python (PYTHONPATH).

DJANGO_SETTINGS_MODULE

Con todos estos cambios en el código, la siguiente parte de este capítulo se centra en las instrucciones específicas para desplegar Django en distintos entornos, tal como Apache, Gunicorn... Las instrucciones son diferentes para cada entorno, pero una cosa es la misma: en cada caso necesitas decirle al servidor Web cuál es tu: DJANGO_SETTINGS_MODULE. Este es el punto de entrada de tu aplicación Django.

DJANGO_SETTINGS_MODULE enlaza tu archivo de configuraciones, el cual apunta a ROOT_URLCONF, que a su vez enlaza tus vistas y así sucesivamente.

El objeto *application*

El concepto clave para implementar usando WSGI es el llamable application que es el modulo que el servidor de aplicaciones utiliza para comunicarse con el código Python. Este comúnmente se provee como un objeto llamado *application* el cual es un modulo Python accesible por el servidor.

El comando startproject crea un archivo llamado **wsgi.py** este contiene un modulo llamable llamado application. Este es usado tanto en el servidor de desarrollo, como en el despliegue para producción, por lo que no necesitas crearlo.

El servidor WSGI obtiene la ruta de la configuración del llamable application. El servidor de desarrollo “runserver” lee la configuración de WSGI_APPLICATION, la cual enlaza a el llamable application en el archivo wsgi.py, lo mismo pasa con un servidor en producción.

Configura el modulo settings

Cuando el servidor WSGI carga tu aplicación, Django necesita importar el modulo de configuraciones settings – que es donde se define completamente la aplicación.

Django usa la variable de entorno DJANGO_SETTINGS_MODULE que contiene la ruta para localizar apropiadamente este modulo. Puedes usar diferentes valores para desarrollo y producción; todo depende de cómo organices tus configuraciones.

Si esta variable no está establecida, el valor por defecto es misitio.settings, donde misitio es el nombre de tu proyecto y settings es el nombre del archivo settings.py. Esta es la forma en que runserver carga el archivo de configuraciones por defecto.

Como desplegar con WSGI

La plataforma dominante de implementación para Django es WSGI, el estándar Python para servidores y aplicaciones Web.

El comando administrativo startproject establece por defecto, un simple archivo de configuración WSGI, el cual puedes personalizarse según las necesidades de tu proyecto, y directamente adaptarse a cualquier servidor que ofrezca soporte para WSGI, como los siguientes servidores:

- mod_wsgi
- apache
- gunicorn
- uwsgi

Usando Django con Apache y mod_wsgi

Desplegar Django con Apache y mod_wsgi es la manera probada y comprobada de usar Django en un servidor en producción.

mod_wsgi es un modulo de Apache que puede hospedar cualquier aplicación Python WSGI, incluyendo Django. El cual trabaja con cualquier versión de Apache que soporte mod_wsgi.

La Documentación oficial es fantástica, y el código fuente incluye detalles sobre el modo de usar mod_wsgi. Por lo que probablemente quieras empezar por leer la documentación sobre instalación y configuración.

Configuración básica

Para configurar Django con mod_wsgi, primero debes asegurarte de que tienes instalado Apache con el módulo mod_wsgi activado. Esto usualmente significa tener una directiva *LoadModule* en tu archivo de configuración de Apache. Parecida a esta:

```
LoadModule wsgi_module  /modules/mod_wsgi.so
```

Una vez que has instalado y activado mod_wsgi, edita el archivo httpd.conf de tu servidor Web Apache y agrega lo siguiente. Si estas usando una versión de Apache anterior a la 2.4, remplaza Require all granted con Allow from all y tambien agrega la línea Order deny,allow arriba de esta.

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonPath /path/to/mysite.com
<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

El primer fragmento de la línea WSGIScriptAlias es la ruta base que quieres servir tu aplicaciones en (/ indica la raíz de la url) y la segunda es la localización de el “archivo WSGI” –ver más abajo en tu sistema, usualmente dentro del paquete de tu proyecto (misitio en este ejemplo.) Esto le dice al servidor Apache “Usa mod_wsgi para cualquier petición URL en ‘/’ o bajo ella, usando la aplicación WSGI definida en ese archivo”.

La línea WSGIPythonPath se asegura que el paquete del proyecto esté disponible para importar la ruta de búsqueda de Python; en otras palabras se asegura que import misitio trabaje.

La pieza <Directory> solo se asegura de que Apache pueda acceder al archivo wsgi.py, ya que se utiliza para apuntar a lugares de nuestra sistema de archivos,
Lo siguiente que necesitas hacer, es asegurarte que exista una archivo wsgi.py, como seguramente te diste cuenta el comando startproject crea este archivo por defecto al crear el proyecto, de otra forma tendrías que crear este archivo manualmente.

☠ **Advertencia:** Si varios sitios están siendo ejecutados en un simple proceso mod_wsgi, todos ellos usarán las configuraciones de cualquiera de los procesos que corra primero. Esto puede se solventado con un pequeño cambio en el archivo wsgi.py (ve los comentarios en el archivo para detalles) o puedes asegurarte de cada sitio sea ejecutado en un proceso independiente, usando su propio demonio, sobrescribiendo el valor por defecto usando: os.environ["DJANGO_SETTINGS_MODULE"] = "misitio.settings" en el arhvio wsgi.py

---

Usando virtualenv

Si has instalado las dependencias Python de tu proyecto dentro de virtualenv, necesitas agregar la ruta de "virtualenv's" al directorio site-packages, así como el camino de búsqueda. Para hacer esto agrega una ruta de búsqueda adicional a la directiva WSGIPythonPath, con las múltiples rutas separadas por dos puntos (:) si estas usando un sistema del tipo UNIX, o punto y coma (;) si estas usando Windows. Si cualquier parte de la ruta al directorio contiene caracteres como espacios, la cadena completa de argumentos para WSGIPythonPath debe ser citada:

```
WSGIPythonPath /path/to/mysite.com:/path/to/your/venv/lib/python3.X/site-packages
```

Asegúrate de darle la ruta correcta a tu virtualenv, y remplazar python3.X con la versión correcta de Python (por ejemplo python3.4).

Usando mod_wsgi en modo demonio

"Modo Demonio" es el modo recomendado para ejecutar mod_wsgi (en plataformas que no son Windows). Para crear el grupo de procesos requeridos por el demonio y delegar la instancia Django para ejecutarse, necesitas agregar apropiadamente las directivas WSGIDaemonProcess y WSGIProcessGroup. Un cambio mas es requerido en la anterior configuración si utilizas el modo demonio no puedes usar WSGIPythonPath, en lugar de eso debes usar la opción python-path para WSGIDaemonProcess, por ejemplo:

```
WSGIDaemonProcess example.com python-
path=/path/to/mysite.com:/path/to/venv/lib/python3.4/site-packages
```

Otra opción para servir los archivos multimedia, que no sea el mismo VirtualHost Apache que usa Django, puedes configurar Apache para que sirva algunas URLs estáticas y otras usando la interface mod_wsgi para Django

Este ejemplo configura Django en la raíz del sitio, pero explícitamente sirve robots.txt, favicon.ico, y cualquier archivo CSS, y cualquier cosa en el espacio de URL

/static/ y /media/ será tratado como archivos estáticos. Todas las demás URLs será servidas usando mod_wsgi:
Alias /robots.txt /path/to/mysite.com/static/robots.txt

```
Alias /robots.txt /path/to/mysite.com/static/robots.txt
Alias /favicon.ico /path/to/mysite.com/static/favicon.ico
AliasMatch ^/([^/]*\.css) /path/to/mysite.com/static/styles/$1
Alias /media/ /path/to/mysite.com/media/
Alias /static/ /path/to/mysite.com/static/
<Directory /path/to/mysite.com/static>
Require all granted
</Directory>
<Directory /path/to/mysite.com/media>
Require all granted
</Directory>
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

Si estas usando una versión de Apache anterior a la 2.4, remplaza Require all granted con Allow from all y tambien agrega la línea Order deny,allow arriba de esta.

Como servir los archivos de la interfaz administrativa

Cuando django.contrib.staticfiles está en el archivo de configuraciones en la variable INSTALLED_APPS El entorno de desarrollo de Django sirve automáticamente los archivos estáticos de la interfaz administrativa (y de cualquier aplicación instalada). Este no es el caso cuando usas otro servidor para este trabajo. Tu eres responsable de configurar Apache o cualquier otro servidor Web que utilices, para servir los archivos de la interfaz administrativa.

Los archivos de la interfaz administrativa se localizan en django/contrib/admin/static/admin en la distribucion de Django Es *fuertemente* recomendable usar django.contrib.staticfiles para manejar los archivos de la interfaz administrativa (Junto con el servidor Web, esto significa que puedes usar el comando collectstatic para recolectar todos los archivos estáticos en STATIC_ROOT y luego configurar el servidor Web para servir STATIC_ROOT, pero aquí estan otras tres formas de hacer lo mismo:

Crea un enlace simbólico a los archivos estáticos de la interfaz administrativa desde la raíz de Apache (esto puede requerir usar +FollowSymLinks en la configuración de Apache)
Usa una directiva Alias, como mostramos arriba, un alias apropiado a la URL (probablemente STATIC_URL + admin/) a la posición actual de los archivos estáticos.

Copia los archivos estáticos de la interfaz administrativa de modo que se localicen dentro de la raíz de Apache.

Como usar Django con Gunicorn

Gunicorn ('Unicornio verde') es un servidor WSGI implementado en Python. No tiene dependencias y es fácil de instalar y usar.

Hay dos formas de usar Gunicorn con Django. La primera es tratar a Gunicorn simplemente como otra aplicación WSGI. La segunda es usar Gunicorn de forma especial integrándolo con Django.

Una vez instalado Gunicorn, tenemos disponible un comando gunicorn que inicia el proceso del servidor Gunicorn. Esto es tan simple, ya que gunicorn solo necesita ser llamado con la localización del objeto aplication del WSGI.:

```
gunicorn [OPTIONS] MODULO
```

Donde MODULO es un patron del tipo NOMBRE_MODULO:NOMBRE_VARIABLE. El nombre del modulo debe ser la ruta completa al proyecto, mientras que el nombre de la variable se refiere al llamable WSGI el cual debe de encontrarse en el modo especificado.

Así para un típico proyecto Django, invocar a gunicorn se vería así:

```
gunicorn misitio.wsgi:application
```

Donde misitio es el nombre del proyecto y application es el llamable del WSGI.

(Esto requiere que el proyecto este en la ruta de búsqueda de Python; la forma más simple de hacer esto, es asegurarse de ejecutar el comando "gunicorn" desde el mismo directorio en que esta el archivo manage.py)

Como usar Django con uWSGI

uWSGI es un servidor rápido codificado en C, autoregenerable y desarrollado como una aplicación amigable para los administradores de sistemas.

Prerrequisitos: Uwsgi

La wiki de WSGI describe algunos método de instalación. Usando pip, el manejador de paquetes python puedes instalar cualquier versión de uWSGI con un simple comando. Por ejemplo:

```
# Instala la  versión estable.
sudo pip install uwsgi

# O installa la LTS (Soporte a largo plazo).
sudo pip install http://projects.unbit.it/downloads/uwsgi-lts.tar.gz
```

Advertencia: Algunas distribuciones, incluidas Debían y Ubuntu, incluyen versiones antiguas de uWSGI que no cumplen las especificaciones WSGI. Versiones anteriores a la 1.2.6 no llaman a close en el objeto de repuesta después de manejar una petición. En estos casos django.core.signals.request_finished no envía las señales. Esto puede dar lugar a conexiones lentas a los servidores de la base de datos y memcache.

WSGI model

uWSGI opera en un modelo cliente-servidor. El servidor web (por ejemplo nginx, Apache) se comunicacn con un proceso del "trabajo" django-uwsgi para servir dinámicamente el contexto. Consulta la documentación a fondo de uWSGI's para obtener más detalles.

Configurar e iniciar el servidor uWSGI

uWSGI soporta múltiples formas para configurar el proceso. Puede iniciar mediante un comando o leyendo un archivo de configuraciones de inicio. Puedes consultar más ejemplos y la documentación de uWSGI para obtener detalles más específicos.

Este es un ejemplo de un comando para iniciar el servidor uWSGI:

```
uwsgi --chdir=/path/to/your/project \
    --module=misitio.wsgi:application \
    --env DJANGO_SETTINGS_MODULE=misitio.settings \
    --master --pidfile=/tmp/project-master.pid \
    --socket=127.0.0.1:49152 \ # puede usar también un archivo.
    --processes=5 \ # número de procesos
    --uid=1000 --gid=2000 \ # si es root, uwsgi puede revocar provilegios
    --harakiri=20 \ # para procesos que toman más de 20 segundos
    --max-requests=5000 \ # número máximo de peticiones
    --vacuum \ # limpia el entorno y sale.
    --home=/path/to/virtual/env \ # ruta opcional para  virtualenv
    --daemonize=/var/log/uwsgi/yourproject.log # procesos en Segundo plano
```

Este asume que estas situado en el nivel superior de un paquete llamado misitio y dentro del existe un modulo wsgi.py, que contienen un objeto application WSGI. No deberías tener ningún problema si ejecutaste el comando django-admin.py startproject misitio, ya que este se encarga de crear la estructura del proyecto por default. Si este archivo no existe necesitas crearlo.

Las opciones específicas de Django son:

- **chdir:** La ruta al directorio, que necesita estar en la ruta de importacion de Python. – es decir el directorio que contiene el paquete misitio
- **module**: El modulo WSGI para usar –probablemente el modulo misitio.wsgi que startproject creo.
- **env**: Debe contener por lo menos DJANGO_SETTINGS_MODULE.
- **home**: Opcionalmente la ruta al proyecto, si estas usando virtualenv.

Ejemplo de un archivo de configuración ini:

```
[uwsgi]
chdir=/path/to/your/project
module=misitio.wsgi:application
master=True
pidfile=/tmp/project-master.pid
vacuum=True
max-requests=5000
```

```
daemonize=/var/log/uwsgi/yourproject.log
```

Consulta el manejo de procesos uWSGI para más información sobre iniciar, detener y recargar los procesos del servidor uWSGI.

Lighttpd

lighttpd es un servidor Web liviano usado habitualmente para servir archivos estáticos. Admite FastCGI en forma nativa y por lo tanto es también una opción ideal para servir tanto páginas estáticas como dinámicas, si tu sitio no tiene necesidades específicas de Apache.

Asegúrate que mod_fastcgi está en tu lista de módulos, en algún lugar después de mod_rewrite y mod_access, pero no antes de mod_accesslog. Probablemente desees también mod_alias, para servir los archivos media de la interfaz administrativa.

Agrega lo siguiente a tu archivo de configuración de lighttpd:

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
                # Use host / port instead of socket for TCP fastcgi
                # "host" => "127.0.0.1",
                # "port" => 3033,
                "socket" => "/home/user/mysite.sock",
                "check-local" => "disable",
        )
    ),
)
alias.url = (
        "/media" => "/home/user/django/contrib/admin/media/",
)
url.rewrite-once = (
        "^(/media.*)$" => "$1",
        "^/favicon\.ico$" => "/media/favicon.ico",
        "^(/.*)$" => "/mysite.fcgi$1",
)
```

Ejecutando Múltiples Sitios Django en Una Instancia lighttpd

lighttpd te permite usar "configuración condicional" para permitir la configuración personalizada para cada host. Para especificar múltiples sitios FastCGI, solo agrega un bloque condicional en torno a tu configuración FastCGI para cada sitio:

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
        server.document-root = "/foo/site1"
        fastcgi.server = (
            ...
```

```
        )
        ...
}
# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
        server.document-root = "/foo/site2"
        fastcgi.server = (
            ...
        )
        ...
}
```

Puedes también ejecutar múltiples instalaciones de Django en el mismo sitio simplemente especificando múltiples entradas en la directiva fastcgi.server. Agrega un host FastCGI para cada una.

Escalamiento

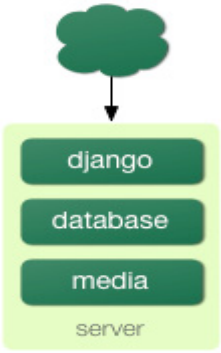
Ahora que sabes cómo configurar Django en un sitio en producción, veamos cómo puedes escalar una instalación Django. Esta sección explica cómo puede escalar un sitio desde un servidor único a un clúster de gran escala que pueda servir millones de peticiones por hora.

Es importante notar, sin embargo, que cada sitio grande es grande de diferentes formas, por lo que escalar es cualquier cosa menos una operación de una solución única para todos los casos. La siguiente cobertura debe ser suficiente para mostrar el principio general, y cuando sea posible, trataremos de señalar donde se puedan elegir distintas opciones.

Primero, haremos una buena presuposición, y hablaremos exclusivamente acerca de escalamiento bajo Apache y wsgi. A pesar de que conocemos vario casos exitosos de desarrollos FastCGI y mod_python medios y grandes, estamos mucho más familiarizados con Apache.

Ejecutando en un Servidor Único

La mayoría de los sitios Web, empiezan ejecutándose en un servidor único, con una arquitectura parecida a la de la Figura 12-1.



Esto funciona bien para sitios pequeños y medianos, y es relativamente barato – puedes instalar un servidor único diseñado para Django por menos de 3,000 dólares.

Imagen 12-1: Ejecutando únicamente un servidor.

Sin embargo, a medida que el tráfico se incremente, caerás rápidamente en *contención de recursos* entre las diferentes piezas de software. Los servidores de base de datos y los servidores Web *adoran* tener el servidor entero para ellos, y cuando corren en el mismo servidor siempre terminan "peleando" por los mismos recursos (RAM, CPU) que prefieren monopolizar.

Esto se resuelve fácilmente moviendo el servidor de base de datos a una segunda máquina, como se explica en la siguiente sección.

Separando el Servidor de Bases de Datos

En lo que tiene que ver con Django, el proceso de separar el servidor de bases de datos es extremadamente sencillo: simplemente necesitas cambiar la configuración de DATABASE_HOST a la IP o nombre DNS de tu servidor. Probablemente sea una buena idea usar la IP si es posible, ya que depender de la DNS para la conexión entre el servidor Web y el servidor de bases de datos no se recomienda.

Con un servidor de base de datos separado, nuestra arquitectura ahora se ve como en la Imagen 12-2.

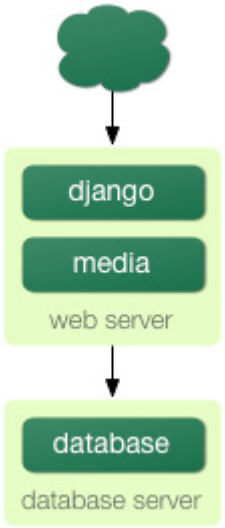


Imagen 12-2: Moviendo la base de datos a un servidor dedicado.

Aquí es donde empezamos a movernos hacia lo que usualmente se llama una arquitectura *n-tier*. No te asustes por la terminología – sólo se refiere al hecho de que diferentes "tiers" de la pila Web separadas en diferentes máquinas físicas.

A esta altura, si anticipas que en algún momento vas a necesitar crecer más allá de un servidor de base de datos único, probablemente sea una buena idea empezar a pensar en pooling de conexiones y/o replicación de bases de datos. Desafortunadamente, no hay suficiente espacio para hacerle justicia a estos temas en este libro, así que vas a necesitar consultar la documentación y/o a la comunidad de tu base de datos para más información.

Ejecutando un Servidor de Medios Separado

Aún tenemos un gran problema por delante, usando únicamente un servidor: servir los archivos media desde la misma caja que maneja el contenido dinámico.

Estas dos actividades tienen su mejor performance bajo distintas circunstancias, y encerrándolas en la misma caja terminarás con que ninguna de las dos tendrá un buen rendimiento. Así que el siguiente paso es separar los medios – esto es, todo lo que *no* es generado por una vista de Django – a un servidor dedicado (ver Imagen 12-3).

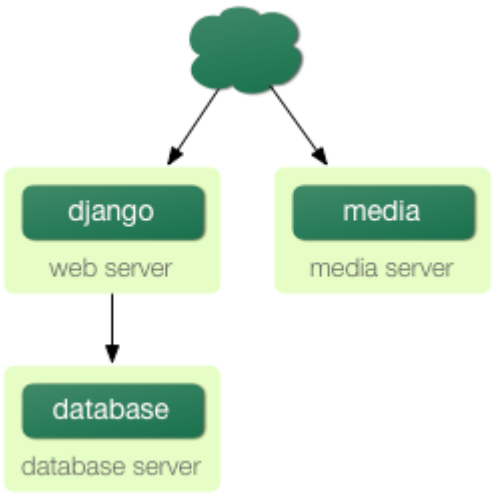


Imagen 12-3: Separando el servidor de medios.

Idealmente, este servidor de medios debería ejecutarse en un servidor Web, optimizado para la entrega de medios estáticos. lighttpd y tux (http://www.djangoproject.com/r/tux/) son dos excelentes elecciones aquí, pero un servidor Apache bien 'personalizado' también puede funcionar.

Para sitios pesados en contenidos estáticos (fotos, videos, etc.), moverse a un servidor de medios separado es doblemente importante y debería ser el *primer* paso en el escalamiento hacia arriba.

De todos modos, este paso puede ser un poco delicado. Si tu aplicación necesita subir archivos Django necesita poder escribir sobre los medios 'subidos' en el servidor de medios. (la configuración de MEDIA_ROOT controla donde escriben estos medios). Si un medio habita en otro servidor, de todas formas necesitas organizar una forma de que esa escritura se pueda hacer a través de la red.

Implementando Balance de Carga y Redundancia

A esta altura, ya hemos separado las cosas todo lo posible.

Esta configuración de tres servidores debería manejar una cantidad muy grande de tráfico – nosotros servimos alrededor de 10 millones de hits por día con una arquitectura de este tipo– así que si creces más allá, necesitarás empezar a agregar redundancia.

En realidad, esto es algo bueno. Una mirada a la Figura 12-3 te permitirá ver que si falla aunque sea uno solo de los servidores, el sitio entero se cae. Así que a medida que agregas servidores redundantes, no sólo incrementas capacidad, sino también confiabilidad.

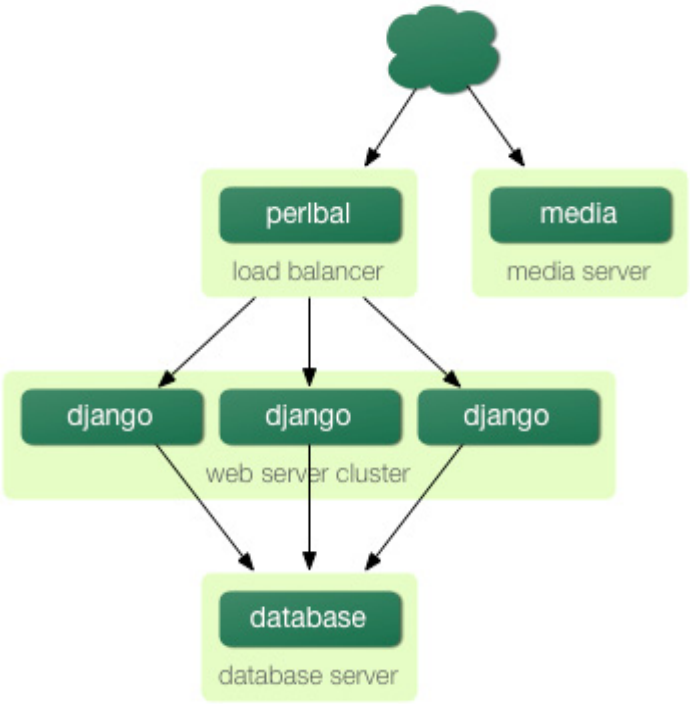


Imagen 12-4: Configuración de un server redundante con balance de carga.

Para este ejemplo, asumamos que el primero que se ve superado en capacidad es el servidor Web. Es fácil tener múltiples copias de un sitio Django ejecutando en diferente hardware. – simplemente copia el código en varias máquinas, e inicia Apache en cada una de ellas.

Sin embargo, necesitas otra pieza de software para distribuir el tráfico entre los servidores: un balanceador de carga. Puedes comprar balanceadores de carga por hardware caro y propietario, pero existen algunos balanceadores de carga por software de alta calidad que son open source.

mod_proxy de Apache es una opción, pero hemos encontrado que Perlbal (http://www.djangoproject.com/r/perlbal/) es simplemente fantástico. Es un balanceador de carga y proxy inverso escrito por las mismas personas que escribieron memcached.

Con los servidores Web en cluster, nuestra arquitectura en evolución empieza a verse más compleja, como se ve en la Imagen 12-4.

Observa que en el diagrama nos referimos a los servidores Web como "el cluster" para indicar que el número de servidores básicamente es variable. Una vez que tienes un balanceador de carga en el frente, puedes agregar y eliminar servidores Web back-end sin perder un segundo fuera de servicio.

Vamos a lo grande

En este punto, los siguientes pasos son derivaciones del último:

A medida que necesites más performance en la base de datos, necesitarás agregar servidores de base de datos replicados. MySQL tiene replicación incorporada; los usuarios de PostgreSQL deberían mirar a Slony (http://www.djangoproject.com/r/slony/) y pgpool (http://www.djangoproject.com/r/pgpool/) para replicación y pooling de conexiones, respectivamente.

Si un solo balanceador de carga no es suficiente, puedes agregar más máquinas balanceadoras de carga y distribuir entre ellas usando DNS round-robin.

Si un servidor único de medios no es suficiente, puedes agregar más servidores de medios y distribuir la carga con tu cluster de balanceadores de carga.

Si necesitas más almacenamiento cache, puedes agregar servidores de cache dedicados.

En cualquier etapa, si un cluster no tiene buena performance, puedes agregar más servidores al cluster.

Después de algunas de estas iteraciones, una arquitectura de gran escala debe verse como en la Imagen 12-5

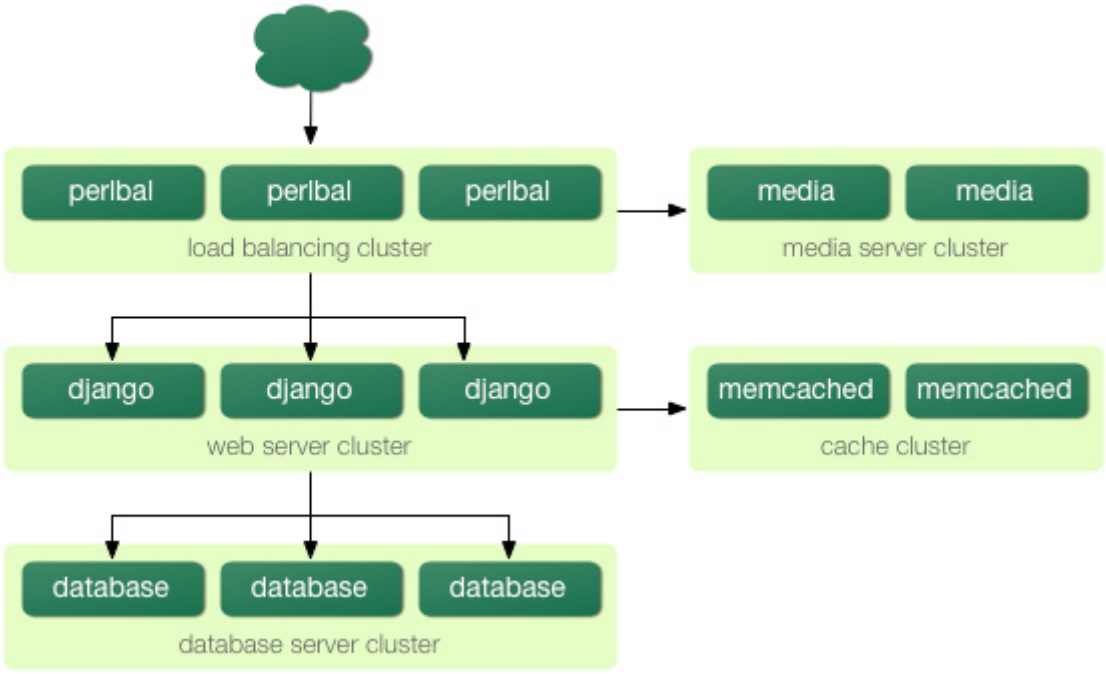


Imagen 12-5. Un ejemplo de configuración de Django de gran escala.

A pesar de que mostramos solo dos o tres servidores en cada nivel, no hay un límite fundamental a cuantos puedes agregar.

Ajuste de Performance

Si tienes grandes cantidades de dinero, simplemente puedes irle arrojando hardware a los problemas de escalado. Para el resto de nosotros, sin embargo, el ajuste de performance es una obligación.

■**Nota:** Incidentalmente, si alguien con monstruosas cantidades de dinero está leyendo este libro, por favor considere hacer una donación sustancial al proyecto Django. Aceptamos diamantes en bruto y lingotes de oro.

Desafortunadamente, el ajuste de performance es más un arte que una ciencia, y es aun más difícil de escribir sobre eso que sobre escalamiento. Si estás pensando seriamente en desplegar una aplicación Django de gran escala, deberás pasar un buen tiempo aprendiendo como ajustar cada pieza de tu stack.

Las siguientes secciones, sin embargo, presentan algunos tips específicos del ajuste de performance de Django que hemos descubierto a través de los años.

No hay tal cosa como demasiada RAM

Incluso la RAM más costosa es relativamente costeable en estos días. Compra toda la RAM que puedas, y después compra un poco más.

Los procesadores más rápidos no mejoran la performance tanto. La mayoría de los servidores Web desperdician el 90% de su tiempo esperando I/O del disco. En cuantos empieces a swappear, la performance directamente se muere. Los discos más rápidos pueden ayudar levemente, pero son mucho más caros que la RAM, así que no cuentan.

Si tienes varios servidores, el primer lugar donde poner tu RAM es en el servidor de base de datos. Si puedes, compra suficiente ram como para tener toda tu base de datos en memoria. Esto no es tan difícil. Hemos diseñado sitios que incluye medio millón de artículos en menos de menos de 2 GB de espacio.

Después, maximiza la RAM de tu servidor Web. La situación ideal es aquella en la que ningún servidor swapea – nunca. Si llegas a ese punto, debes poder manejar la mayor parte del tráfico normal.

Deshabilita Keep-Alive

Keep-Alive es una característica de HTTP que permite que múltiples pedidos HTTP sean servidos sobre una conexión TCP única, evitando la sobrecarga de conectar y desconectar.

Esto parece bueno a primera vista, pero puede asesinar al performance de un sitio Django. Si estás sirviendo medios desde un servidor separado, cada usuario que esté navegando tu sitio solo requerirá una página del servidor Django cada diez segundos aproximadamente. Esto deja a los servidores HTTP esperando el siguiente pedido keep-alive, y un servidor HTTP ocioso consume RAM que debería estar usando un servidor activo.

Usa memcached

A pesar de que Django admite varios back-ends de cache diferentes, ninguno de ellos siquiera se acerca a ser tan rápido como memcached. Si tienes un sitio con tráfico alto, ni pierdas tiempo con los otros –- usa directamente memcached.

Usa memcached siempre

Por supuesto, seleccionar memcached no te hace mejor si no lo usas realmente. El *capítulo* te dice cómo usarlo: aprende a usar el framework de cache de Django, y úsalo en todas las partes que te sea posible. Un uso de cache agresivo y preemptico es usualmente lo único que se puede hacer para mantener un sitio Web funcionando bajo el mayor tráfico.

Únete a la Conversación

Cada pieza del stack de Django – desde Linux a Apache a PostgreSQL o MySQL – tiene una comunidad maravillosa detrás. Si realmente quieres obtener ese último 1% de tus servidores, únete a las comunidades open source que están detrás de tu software y pide ayuda. La mayoría de los miembros de la comunidad del software libre estarán felices de ayudar.

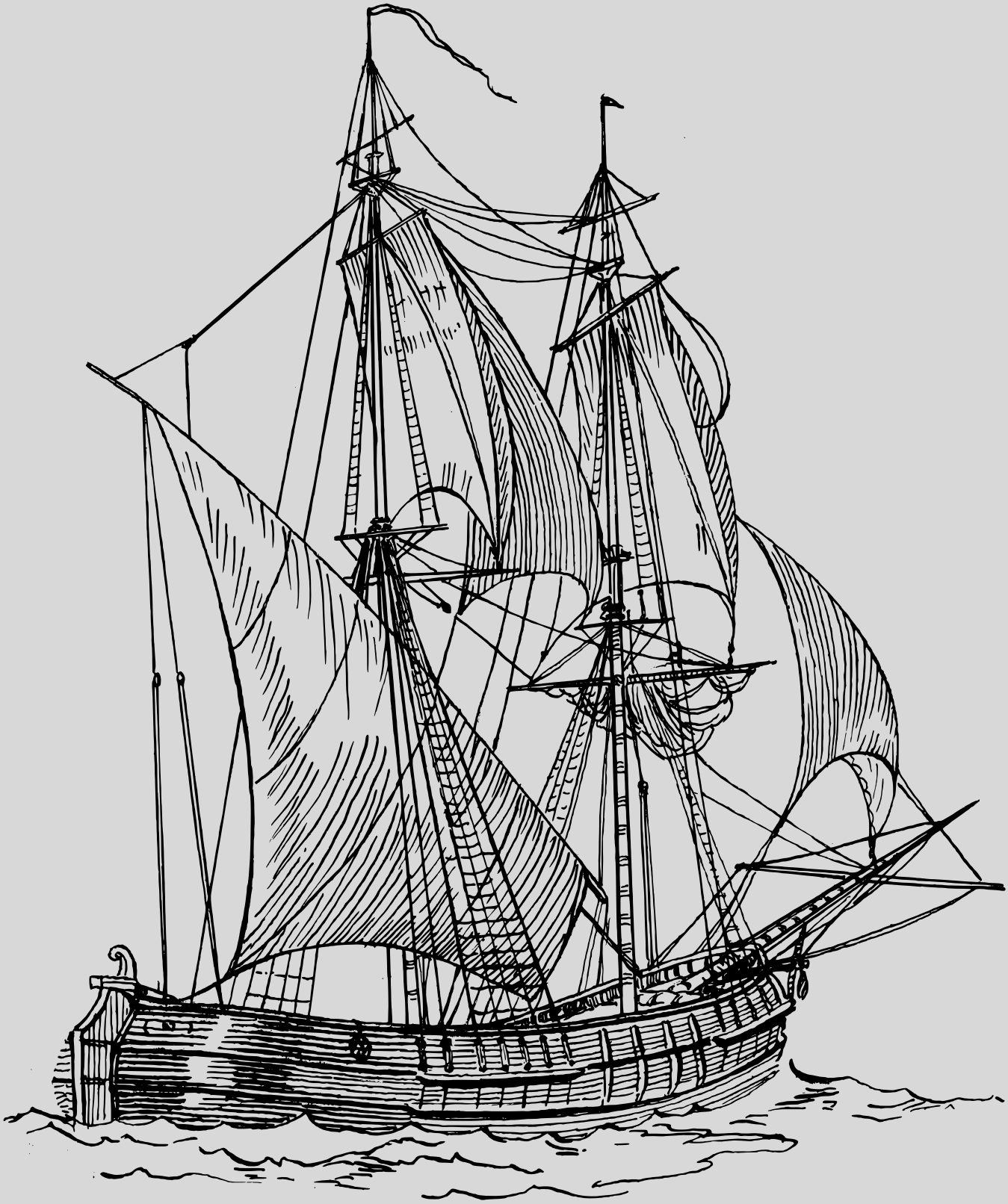
Y también asegúrate de unirte a la comunidad Django. Tus humildes autores son solo dos miembros de un grupo increíblemente activo y creciente de desarrolladores Django. Nuestra comunidad tiene una enorme cantidad de experiencia colectiva para ofrecer.

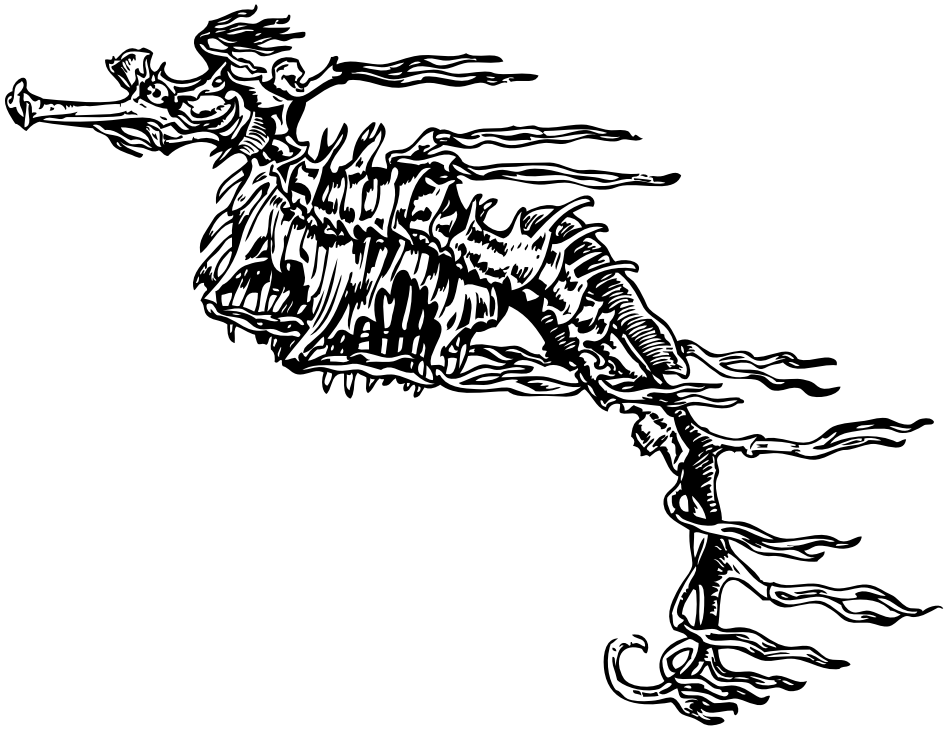
¿Qué sigue?

El resto de los capítulos se enfocan en otras características de Django, que puedes o no necesitar, dependiendo de tus aplicaciones. Siéntete libre de leerlos en el orden que prefieras.

PARTE 3

Baterias incluidas





CAPÍTULO 13

Generación de contenido no HTML

Usualmente cuando hablamos sobre desarrollo de sitios Web, hablamos de producir HTML. Por supuesto, hay mucho más que contenido HTML en la Web; la usamos para distribuir datos en todo tipo de formatos: rss, cvs, pdf's, imágenes, videos y así sucesivamente.

Hasta ahora nos hemos concentrado en el caso común de la producción de HTML, pero en ese capítulo tomaremos un ligero desvío y veremos cómo usar Django para producir otro tipo de contenido.

Django posee varias herramientas útiles que puedes usar para producir algunos tipos comunes de contenido no HTML:

- Feeds de sindicación RSS/Atom
- Mapas de sitios haciendo uso de Sitemaps (un formato XML originalmente desarrollado por Google que provee de ayuda a motores de búsqueda)

Examinaremos cada una de esas herramientas un poco más adelante, pero antes cubriremos los principios básicos.

Lo básico: Vistas y tipos MIME

¿Recuerdas esto del *capítulo 3*?

> "Una función vista, o una vista por abreviar, es simplemente una función en Python que recibe una petición Web y retorna una respuesta Web. Esta respuesta puede ser el contenido HTML de una página Web, una redirección, un error 404, un documento XML, una imagen... en realidad, cualquier cosa".

Más formalmente, una función *vista debe*

- Aceptar una instancia HttpRequest como primer argumento.
- Retornar una instancia HttpRequestsponse como respuesta.

La clave para retornar contenido no HTML desde una vista reside en la clase HttpResponse, específicamente en el argumento mimetype del constructor. Cambiando el tipo MIME, podemos indicarle al navegador que hemos retornado una respuesta en un formato diferente.

Por ejemplo, veamos una vista que devuelve una imagen PNG. Para mantener las cosas sencillas, simplemente leeremos un fichero desde el disco:

```
from django.http import HttpResponse

def mi_imagen(request):
    datos_imagen = open("/ruta/a/imagen.png", "rb").read()
    return HttpResponse(datos_imagen, mimetype="imagen/png")
```

¡Eso es todo! Si sustituimos la ruta de la imagen en la llamada a open() con la ruta a una imagen real, podemos usar esta vista bastante sencilla para servir una imagen, y el navegador la mostrará correctamente.

La otra cosa importante a tener presente es que los objetos HttpResponse implementan el API estándar de Python para ficheros. Esto significa que podemos usar una instancia de HttpResponse en cualquier lugar donde Python (o biblioteca de terceros) espera un fichero.

Como un ejemplo de cómo funciona esto, veamos la producción de CSV con Django.

Producción de CSV

CSV es un formato de datos sencillo que suele ser usada por programas de hojas de cálculo. Básicamente es una serie de filas en una tabla, cada celda en la fila está separada por comas (CSV significa *comma-separated values*). Por ejemplo, aquí tienes una lista de pasajeros "problemáticos" en líneas aéreas en formato CSV:

```
Año, Pasajeros problemáticos
1995,146
1996,184
1997,235
1998,200
1999,226
2000,251
2001,299
2002,273
2003,281
2004,304
2005,203
```

El listado anterior, contiene números reales; cortesía de la Administración Federal de Aviación (FAA) de E.E.U.U. Tomados de:
http://www.faa.gov/data_statistics/passengers_cargo/unruly_passengers/.

Aunque CSV parezca simple, no es un formato que ha sido definido formalmente. Diferentes programas producen y consumen diferentes variantes de CSV, haciendo un poco complicado usarlo. Afortunadamente, Python incluye una biblioteca estándar para CSV, llamada csv; que es bastante robusta.

Debido a que el módulo csv opera sobre objetos de forma similar ha como lo hace con archivos, es muy fácil usar un HttpResponse en lugar de un fichero:

```
import csv
from django.http import HttpResponse

    # Numero de pasajeros problemáticos en el periodo 1995-2005. En una
    # aplicacion real estos datos probablemente vendrían de una base de datos
    # o de algún otro tipo de almacenamiento externo.
PASAJEROS_PROBLEMATICOS = [146,184,235,200,226,251,299,273,281,304,203]

def pasajeros_problematicos_csv(request):
    # Crea un objeto HttpResponse con las cabeceras del CVS correctas.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename=problematicos.csv'

    # Crea el escritor CSV usando un HttpResponse como "archivo."
    writer = csv.writer(response)
    writer.writerow(['Año', 'Pasajeros problemáticos en aerolínea'])
    for (year, num) in zip(range(1995, 2006), PASAJEROS_PROBLEMATICOS):
        writer.writerow([year, num])

    return response
```

- El código y los comentarios deberían ser bastante claros, pero hay unas pocas cosas que merecen mención especial:

- Se le da a la respuesta el tipo MIME text/csv (en lugar del tipo predeterminado text/html). Esto le dice a los navegadores que el documento es un fichero CSV.

- La respuesta obtiene una cabecera Content-Disposition adicional, la cual contiene el nombre del fichero CSV. Esta cabecera (bueno, la parte "adjunta") le indicará al navegador que solicite la ubicación donde guardará el fichero (en lugar de simplemente mostrarlo). El nombre de fichero es arbitrario; llámalo como quieras. Será usado por los navegadores en el cuadro de diálogo "Guardar como..."

- Usar el API de generación de CSV es sencillo: basta pasar response como primer argumento a csv.writer. La función csv.writer espera un "objeto de tipo fichero", y los objetos de tipo HttpResponse se ajustan a ello.

- Por cada fila en el fichero CSV, invocamos a writer.writerow, pasándole un objeto iterable como una lista o una tupla.

- El módulo CSV se encarga de poner comillas por ti, así que no tendrás que preocuparte por *escapar* caracteres en las cadenas que tengan comillas o comas en su interior. Limítate a pasar la información a writerow(), que hará lo correcto.

Este es el patrón general que usarás siempre que necesites retornar contenido no HTML: crear un objeto HttpResponse de respuesta (con un tipo MIME especial), pasárselo a algo que espera un fichero, y luego devolver la respuesta.

Veamos unos cuantos ejemplos más.

Generar PDF's en Django

El Formato Portable de Documentos (PDF, por Portable Document Format) es un formato desarrollado por Adobe que es usado para representar documentos imprimibles, completos con formato perfecto hasta un nivel de detalle medido en pixels, tipografías empotradas y gráficos de vectores en 2D. Puedes pensar en un documento PDF como el equivalente digital de un documento impreso; efectivamente, los PDF's se usan normalmente cuando se necesita entregar un documento a alguien para que lo imprima.

Puedes generar PDF's fácilmente con Python y Django gracias a la excelente biblioteca open source ReportLab (🌎 http://www.reportlab.org/rl_toolkit.html). La ventaja de generar ficheros PDF's dinámicamente es que puedes crear PDF's a medida para diferentes propósitos – supongamos, para diferentes usuarios u diferentes contenidos.

Por ejemplo, hemos usado Django y ReportLab en KUSports.com para generar programas de torneos de la NCAA personalizados, listos para ser impresos.

Instalar ReportLab

Antes de que puedas generar algún PDF, necesitas instalar ReportLab. Esto es usualmente muy simple: sólo descarga e instala la biblioteca desde 🌎 http://www.reportlab.org/downloads.html.

La guía del usuario (naturalmente disponible en formato PDF) la puedes encontrar en 🌎 http://www.reportlab.org/rsrc/userguide.pdf, contiene instrucciones de instalación adicionales.

Si estás usando una distribución moderna de Linux, podrías desear comprobar con la utilidad de manejo de paquetes de software antes de instalar ReportLab. La mayoría de los repositorios de paquetes ya incluyen ReportLab. Aunque simplemente Puedes utilizar pip install reportlab.

Prueba tu instalación importando la misma en el intérprete interactivo Python:

```
>>> import reportlab
>>>
```

Si ese comando no lanza ningún error, la instalación funcionó.

Vistas que producen PDF's en Django

Del mismo modo que CSV, la generación de PDF's en forma dinámica con Django es sencilla porque la API ReportLab actúa sobre objetos de forma similar a como lo hace con ficheros (*file-like* según la jerga Python).

A continuación un ejemplo “Hola Mundo”:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse
```

```
def hola_pdf(request):
    # Crea un objeto HttpResponse con las cabeceras del PDF correctas.
    response = HttpResponse(content_type='application/pdf')
    # Abre el PDF en la ventana del navegador
    #response['Content-Disposition'] = 'filename="archivo.pdf"'
    # inicia el cuadro de dialogo "abrir con:"
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'
    # Crea el archivo PDF, usando el objeto response como un "archivo".
    p = canvas.Canvas(response)
    # Dibuja cosas en el PDF. Aqui se genera el PDF. Consulta la
    # documentación para obtener una lista completa de funcionalidades.
    p.roundRect(0, 750, 694, 120, 20, stroke=0, fill=1)
    p.setFillColorRGB(0,1,0)
    # La fuente y el tamaño
    p.setFont('Times-Bold',28)
    p.drawString(50, 800, "Bienvenidos a Django")
    p.setFont('Times-Roman', 12)
    p.drawString(250, 780, "Hola mundo")
    p.setFont('Times-Bold',150)
    p.setFillColorRGB(0,0,0)
    p.drawString(70, 400, "Django")
    # Cierra el objeto PDF limpiamente y termina.
    p.showPage()
    p.save()

    return response
```

El código y los comentarios deberían explicarse por sí mismos, pero son necesarias algunas notas adicionales:

Usamos el tipo MIME application/pdf. Esto le indica al navegador que el documento es un fichero PDF y no un fichero HTML. Si no incluyes esta información, los navegadores web probablemente interpretarán la respuesta como HTML, lo que resultará en jeroglíficos en la ventana del navegador.

La respuesta obtiene una cabecera Content-Disposition adicional, la cual contiene el nombre del archivo PDF. Este nombre es arbitrario: llámalo como quieras. Solo será usado para abrir el cuadro de dialogo en el navegador "Guardar como..."

En el ejemplo le agregamos attachment a la respuesta de la cabecera Content Disposition al nombre del archivo. Esto fuerza a los navegadores Web a presentar una ventana de diálogo/confirmación para manipular el documento por defecto usando un programa externo, sin embargo si dejamos en blanco attachment el navegador manipulará el PDF usando cualquier plugin que haya sido configurada para manejar este tipo de archivos dentro del navegador, el código es el siguiente:

```
response['Content-Disposition'] = 'filename="archivo.pdf"'
```

Interactuar con la API ReportLab es sencillo: sólo pasa response como el primer argumento a canvas.Canvas. La clase Canvas espera un objeto tipo archivo, por lo que los objetos HttpResponse se ajustarán a la norma.

Todos los métodos de generación de PDF subsecuentes son llamados pasándoles el objeto PDF (en este caso p), no response.

Finalmente, es importante llamar a los métodos showPage() y save() del objeto PDF (de otra manera obtendrás un fichero PDF corrupto).

PDF's más complejos

Si estás creando un documento PDF complejo considera usar la biblioteca io como un lugar de almacenamiento temporal para tu fichero PDF. Esta biblioteca provee una interfaz para tratar con archivos tipo objetos muy eficientemente.

En el siguiente ejemplo, obtenemos datos directamente de la base de datos, que creamos en los capítulos anteriores y los usamos para crear un PDF sencillo usando el modulo io de Python.

biblioteca/views.py

```
from io import BytesIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse
from biblioteca.models import Libro

def convertir_pdf(request, pk):
    # Obtenemos un queryset, para un determinado libro usando pk.
    try:
        libro = Libro.objects.get(id=pk)
    except ValueError: # Si no existe llamamos a "pagina no encontrada".
        raise Http404()
    # Creamos un objeto HttpResponse con las cabeceras del PDF correctas.
    response = HttpResponse(content_type='application/pdf')
    # Nos aseguramos que el navegador lo abra directamente.
    response['Content-Disposition'] = 'filename="archivo.pdf"'
    buffer = BytesIO()
    # Creamos el objeto PDF, usando el objeto BytesIO como si fuera un "archivo".
    p = canvas.Canvas(buffer)
    # Dibujamos cosas en el PDF. Aquí se genera el PDF.
    # Consulta la documentación para una lista completa de funcionalidades.
    p.roundRect(0, 750, 694, 120, 20, stroke=0, fill=1)
    p.setFont('Times-Bold',32)
    p.setFillColorRGB(1,1,1)
    p.drawString(100, 800, str(libro.titulo))#Obtenemos el titulo de un libro y la portada.
    p.drawImage(str(libro.portada.url), 100, 100, width=400, height=600)
    # mostramos y guardamos el objeto PDF.
    p.showPage()
    p.save()
    # Traemos el valor del bufer BytesIO y devolvemos la respuesta.
    pdf = buffer.getvalue()
    # Cerramos el bufer
    buffer.close()
    response.write(pdf)
    return response
```

Ahora solo enlazamos la vista a la ULRconf así:

Urls.py

```
from django.conf.urls import url
from biblioteca import views
```

```
urlpatterns =[
      url(r'^convertir-pdf/(?P<pk>[0-9]+)/$', views.convertir_pdf, name='convertir-pdf'),
]
```

Otras posibilidades

Hay infinidad de otros tipos de contenido que puedes generar en Python. Aquí tenemos algunas otras ideas y las bibliotecas que podrías usar para implementarlas:

- **Archivos ZIP**: La biblioteca estándar de Python contiene el módulo zipfile, que puede escribir y leer ficheros comprimidos en formato ZIP. Puedes usarla para guardar ficheros bajo demanda, o quizás comprimir grandes documentos cuando lo requieran. De la misma manera puedes generar ficheros en formato TAR usando el módulo de la biblioteca estándar tarfile.

- **Imágenes Dinámicas**: Biblioteca Python de procesamiento de Imágenes (Python Imaging Library, PIL; http://www.pythonware.com/products/pil/) es una herramienta fantástica para producir imágenes (PNG, JPEG, GIF, y muchas más). Puedes usarla para escalar automáticamente imágenes para generar miniaturas, agrupar varias imágenes en un solo marco e incluso realizar procesamiento de imágenes directamente en la web.

- **Ploteos y Gráficos**: Existe un número importante de increíblemente potentes bibliotecas de Python para Ploteo y Gráficos, que se pueden utilizar para generar mapas, dibujos, ploteos y gráficos. Es imposible listar todas las bibliotecas, así que resaltamos algunas de ellas:

  - **matplotlib** (http://matplotlib.sourceforge.net/) puede usarse para generar ploteos de alta calidad al estilo de los generados con MatLab o Mathematica.

  - **pygraphviz** (https://networkx.lanl.gov/wiki/pygraphviz), una interfaz con la herramienta Graphviz (http://graphviz.org/), puede usarse para generar diagramas estructurados de grafos y redes.

En general, cualquier biblioteca Python capaz de escribir en un fichero puede ser utilizada dentro de Django. Las posibilidades son realmente interminables.
Ahora que hemos visto lo básico de generar contenido no-HTML, avancemos al siguiente nivel de abstracción. Django incluye algunas herramientas agradables e ingeniosas para generar cierto tipo de contenido no-HTML.

El Framework de Feeds de Sindicación

Django incluye un framework para la generación y sindicación de *feeds* de alto nivel que permite crear feeds RSS y Atom de manera sencilla.

Nota: RSS y Atom son formatos basados en XML que se puede utilizar para actualizar automáticamente los "feeds" con el contenido de tu sitio. Lee más sobre RSS en http://www.whatisrss.com/, y obtén información sobre Atom en http://www.atomenabled.org/.
Para crear cualquier feed de sindicación, todo lo que necesitas hacer es escribir una pequeña clase Python. Puedes crear tantos feeds como desees.

El framework de generación de feeds de alto nivel es una vista enganchada a /feeds/ por convención. Django usa el final de la URL (todo lo que este después de /feeds/) para determinar qué feed retornar.

Para crear un feed, necesitas escribir una clase Feed y hacer referencia a la misma en tu URLconf (Consulta los capítulos 3 y 8 para más información sobre URLconfs).

Inicialización

Para activar los feeds de sindicación en tu sitio Django, agrega lo siguiente en tu URLconf:

```
from biblioteca.feeds import UltimosLibros

urlpatterns =[
   url(r'^feeds/$', UltimosLibros()),
]
```

Esa línea le indica a Django que use el framework RSS para captar las URLs que comienzan con "feeds/". (Puedes cambiar "feeds/" por algo que se adapte mejor a tus necesidades).

Debes tener en cuenta que:

- El feed es representado por la clase UltimosLibros el cual por convención y claridad residirá en un nuevo archivo llamado feeds.py, en el mismo nivel que models.py, aunque puede residir en cualquier parte del árbol de código.
- La clase Feed debe ser una subclase de django.contrib.syndication.feeds.Feed.
- Una vez que este configurada la URL, necesitas definir la propia clase Feed. Puedes pensar en una clase Feed como un tipo de clase genérica.

Una clase Feed es una simple clase Python que representa un feed de sindicación. Un feed puede ser simple (p. ej. “noticias del sitio”, o una lista de las últimas entradas del blog) o más complejo (p. ej. mostrar todas las entradas de un blog en una categoría en particular, donde la categoría es variable).

Un Feed simple

Siguiendo con el modelo creado en los capítulos anteriores, veamos ahora como crear un simple feed, que muestre los últimos cinco libros agregados a nuestra aplicación biblioteca.

Empecemos por escribir la clase:

biblioteca/feeds.py

```
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse
from django.utils import feedgenerator

from biblioteca.models import Libro
```

```
class UltimosLibrosFeed(Feed):
    # FEED TYPE -- Opcional. Este debe ser una subclase de la clase
    # django.utils.feedgenerator.SyndicationFeed. Este designa
    # el tipo de feed a usar: RSS 2.0, Atom 1.0, etc. Si no se
    # especifica el tipo de feed (feed_type), se asumirá que el tipo
    # es RSS 2.0. Este debe aparecer en una clase, no en una instancia de
    # una clase.

    feed_type = feedgenerator.Rss201rev2Feed

    title = "Feed libros publicados"
    link = "/ultimos-libros/"
    description = "Ultimos libros publicados en la biblioteca digital."

    def items(self):
        """
        Retorna una lista de items para publicar en este feed.
        """
        return Libro.objects.order_by('-fecha_publicacion')[:5]

    def item_title(self, item):
        """
        Toma un item, retornado por el método items(), y devuelve los item's
        del título como cadena normales Python.
        """
        return item.titulo

    def item_description(self, item):
        """
        Toma un item, retornado por el método items(), y devuelve los item's
        con una descripción en forma de cadena normal de Python.
        """
        return item.descripcion

    def item_link(self, item):
        """
        Toma un item, retornado por el método items(), y devuelve la URL de
        los item's. Es usado solo si el modelo no tiene un método
        get_absolute_url() definido.
        """
        return reverse('detalle-libro', args=[item.pk])

    def item_enclosure_url(self, item):
        """
        Toma un item, retornado por el método items(), y devuelve los item's
        adjuntos en la URL.
        """
        return item.portada.url

    def item_enclosure_length(self, item):
        """
        Toma un item, retornado por el método items(), y devuelve el largo
        de los item's adjuntos.
        """
        return item.portada.size
```

```
    item_enclosure_mime_type = "image/jpeg" # Definimos manualmente el tipo MIME.
```

Para conectar la URL con el feed, usamos una instancia de un objeto Feed en la URLconf. Por ejemplo:

urls.py

```
from django.conf.urls import url
from biblioteca.feeds import UltimosLibrosFeed

urlpatterns = [
    # ...
    url(r'^feeds/$', UltimosLibrosFeed()),
    # ...
]
```

Las cosas importantes a tener en cuenta son:

- La clase Feed es una subclase de django.contrib.syndication.views.Feed.

- **title, link,** y **description** corresponden a los elementos RSS estándar <title>, <link>, y <description> respectivamente.

- **items()** es simplemente un método que retorna una lista de objetos que deben incluirse en el feed como elementos <item>. Aunque este ejemplo retorna objetos NewsItem usando la API de base de datos de Django, no es un requerimiento que items() deba retornar instancias de modelos.
- Obtienes unos pocos bits de funcionalidad "gratis" usando los modelos de Django, pero items() puede retornar cualquier tipo de objeto que desees.

Hay solamente un paso más. En un feed RSS, cada <item> posee <title>, <link>, y <description>. Por lo que es necesario decirle al framework qué datos debe poner en cada uno de los elementos.

- Para especificar el contenido de <title> y <description>, Django trata de llamar a los metodos item_title() e item_description() en la clase Feed. Estos son pasados como simples parámetros item, el cual es el objeto en sí mismo. También estos metodos son opcionales; por defecto la representación Unicode del objeto es usado en ambos.

- Para especificar contenido con algún formato en especifico para <title> y <description>, crea plantillas Django (ver *capítulo 4*) Puedes especificar la ruta con los atributos title_template y description_template en la clase Feed. El sistema RSS rende riza dicha plantilla por cada ítem, pasándole dos variables de contexto para plantillas:

  {{ obj }}: El objeto actual (uno de los tantos que retorna en items().
  {{ site }}: Un objeto django.models.core.sites.Site representa el sitio actual. Esto es útil para *{{ site.domain }}* o *{{ site.name }}.*

  Si no creas una plantilla para el título o la descripción, el framework utilizará la plantilla por omisión "{{ obj }}" – exacto, la cadena normal de representación del objeto.

También puedes cambiar los nombres de estas plantillas especificando title_template y description_template como atributos de tu clase Feed.

- Para especificar el contenido de <link>, hay dos opciones. Por cada ítem en items(), Django primero tratará de ejecutar el método get_absolute_url() en dicho objeto. Si dicho método no existe, entonces trata de llamar al método item_link() en la clase Feed, pasándole un único parámetro, item, que es el objeto en sí mismo.

Ambos metodos: get_absolute_url() y item_link() deben retornar la URL del ítem como una cadena normal de Python.

También es posible pasarle información adicional a title y a description en las plantillas, si es que necesitas suministrar más información a las dos variables anteriores. Para hacerlo solo necesitas implementar el método get_context_data en la subclase Feed.

Por ejemplo:

```
from django.contrib.syndication.views import Feed
from biblioteca.models import Libro

class UltimosLibrosFeed(Feed):
    # NOMBRES PLANTILLAS -- Opcionales. Estas deben de ser cadenas de texto
    # que representan el nombre de las plantillas que Django usara para
    # renderizar el titulo y la descripción del los items del Feed.
    # Ambos son opcionales. Si no se especifica una plantilla, se usara
    # el método item_title() o item_description() en su lugar.

    title = "Mis Libros" # Hard-coded titulo.
    link= "/libros/"
    description_template = "feeds/libros.html" # La plantilla

    def items(self):
        """
        Retorna una lista de items para publicar en este feed.
        """
        return Libro.objects.order_by('-fecha_publicacion')[:5]

    def get_context_data(self, **kwargs):
        """
        Toma la petición actual y los argumentos de la URL, y
        devuelve un objeto que representa este feed. Levanta una
        excepción del tipo django.core.exceptions.ObjectDoesNotExist
        si existe algún error.
        """
        context = super(UltimosLibrosFeed, self).get_context_data(**kwargs)
        context['foo'] = 'bar'

        return context
```

Y en la plantilla algo como esto:

```
{{ foo }}: {{ obj.description }}
```

Este método será llamado una vez por cada item en la lista de libros devuelta por items() con los siguientes argumentos clave:

1. **item** El actual item. Por razones de compatibilidad, el nombre de esta variable de contexto es {{ obj }}.

2. **obj** El objeto devuelto por el método get_object(). Por defecto este no es expuesto en las plantillas para evitar confusión con get_object(). (ver arriba), pero se puede usar en la implementacion de el método get_context_data().

3. **site** El sitio actual, descrito anteriormente.

4. **request** La petición actual o request.

Como puedes ver el comportamiento de get_context_data() es muy similar al de las vistas genéricas - solo llamas a la super clase() para extraer datos del contexto de la clase padre, agregas datos y devuelves el diccionario modificado.

Un Feed más complejo

El framework también permite la creación de feeds más complejos mediante el uso de parámetros.

Por ejemplo, http://chicagocrime.org ofrece un feed RSS de los crímenes recientes de cada departamento de policía en Chicago. Sería tonto crear una clase Feed separada por cada departamento; esto puede violar el principio “No te repitas a ti mismo” (DRY, por “Do not repeat yourself”) y crearía acoplamiento entre los datos y la lógica de programación.

En su lugar, el framework de feeds de sindicación te permite crear feeds genéricos que retornan items basados en la información de la URL del feed.

En chicagocrime.org, los feed por departamento de policía son accesibles mediante URLs como estas:

- /beats/613/rss/ : Retorna los crímenes más recientes para el departamento 0613

- /beats/1424/rss/: Retorna los crímenes más recientes para el departamento 1424

Estas funcionan con una URLconf parecida a esta:

```
url(r'^beats/(?P<beat_id>[0-9]+)/rss/$', BeatFeed()),
```

El slug aquí es "beats". El framework de sindicación ve las partes extra en la URL tras el slug – 0613 y 1424 – y te provee un gancho (*hook*) para que le indiques qué significa cada uno de esas partes y cómo influyen en los items que serán publicados en el feed.

Tal como en una vista, los argumentos en la URL son pasados mediante el método get_object() junto con el objeto de la petición.

Un ejemplo aclarará esto. Este es el código para los feeds por departamento:

```
from django.contrib.syndication.views import FeedDoesNotExist
from django.shortcuts import get_object_or_404
```

```
class BeatFeed(Feed):
    description_template = 'feeds/beat_description.html'

    def get_object(self, request, beat_id):
        return get_object_or_404(Beat, pk=beat_id)

    def title(self, obj):
        return "Police beat central: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        return Crime.objects.filter(beat=obj).order_by('-crime_date')[:30]
```

Para generar los feed's <title>, <link> y <description>, Django usa los metodos title(), link() y description(). En el ejemplo anterior, estos eran atributos simples de una clase, pero este ejemplo ilustra que estos pueden ser tanto métodos o cadenas. Por cada title, link y description,

Django sigue el siguiente algoritmo.

1. Primero trata de llamar al método, pasando el argumento obj, donde obj es el objeto retornado por get_object().

2. Si eso falla, trata de llamar al método sin argumentos.

3. Si eso falla, usa los atributos de clase.

Nota que items() en el ejemplo anterior, también toma como argumento a obj. El algoritmo para items es el mismo que se describe en el paso anterior – primero prueba items(obj), después items(), y finalmente un atributo de clase items (que debe ser una lista).

Estamos usando una plantilla muy simple para las descripciones de los ìtems, como esta:

```
{{ obj.description }}
```

Especificar el tipo de Feed

Por omisión, el framework de feeds de sindicación produce RSS 2.0. Para cambiar eso, agrega un atributo feed_type a la clase Feed:

```
from django.utils.feedgenerator import Atom1Feed

class MiFeed(Feed):
    feed_type = Atom1Feed
```

Observa que asignas como valor de feed_type una clase, no una instancia. Los tipos de feeds disponibles actualmente se muestran en la siguiente tabla.

Clase Feed	Formato
django.utils.feedgenerator.Rss201rev2Feed	RSS 2.01 (por defecto)
django.utils.feedgenerator.RssUserland091Feed	RSS 0.91
django.utils.feedgenerator.Atom1Feed	Atom 1.0

Tabla 13-1. Tipos de Feeds disponibles en Django.

Adjuntos

Para especificar archivos adjuntos o *enclosures* (p. ej. recursos multimedia asociados al ítem del feed tales como feeds de podcasts MP3, imagenes), usa los métodos item_enclosure_url, item_enclosure_length, e item_enclosure_mime_type.

Por ejemplo:

biblioteca/feeds.py

```
from django.contrib.syndication.views import Feed
from biblioteca.models import Libro

class UltimosLibrosConAdjuntos(Feed):
    title = "Ultimas portadas de Libros"
    link = "/feeds/ejemplo-con-adjuntos/"

    def items(self):
        return Libro.objects.all()[:30]

    def item_enclosure_url(self, item):
        return item.portada.url

    def item_enclosure_length(self, item):
        return item.portada.size

    item_enclosure_mime_type = "image/jpeg" # Definimos un mime-type
```

Esto asume, por supuesto que estamos usando el modelo Libro el cual contiene un campo llamado portada (que es una imagen), al cual se accede a su URL mediante portada.url y mediante portada.size obtenemos el tamaño en bytes.

Y la url queda como sigue:

urls.py

```
from django.conf.urls import url

from biblioteca.feeds import UltimosLibrosConAdjuntos

urlpatterns = [
    # ...
```

```
    url(r'^feeds/adjuntos/$', UltimosLibrosConAdjuntos()),
    # ...
]
```

Lenguage

Los Feeds creados por el framework de sindicación incluyen automáticamente la etiqueta <language> (RSS 2.0) o el atributo xml:lang apropiados (Atom). Esto viene directamente de tu variable de configuración LANGUAGE_CODE.

URLs

El método/atributo link puede retornar tanto una URL absoluta (p. ej. "/blog/") como una URL con el nombre completo de dominio y protocolo (p. ej. "http://www.example.com/blog/"). Si link no retorna el dominio, el framework de sindicación insertará el dominio del sitio actual, acorde a la variable de configuración SITE_ID.

Los feeds Atom requieren un <link rel="self"> que define la ubicación actual del feed. El framework de sindicación completa esto automáticamente, usando el dominio del sitio actual acorde a la variable de configuración SITE_ID.

Publicar feeds Atom y RSS conjuntamente

Algunos desarrolladores prefieren ofrecer ambas versiones Atom *y* RSS de sus feeds. Esto es simple de hacer con Django: solamente crea una subclase de tu clase feed y asigna a feed_type un valor diferente. Luego actualiza tu URLconf para agregar una versión extra. Aquí un ejemplo usando completo:

biblioteca/feeds.py

```
from django.contrib.syndication.views import Feed
from django.utils.feedgenerator import Atom1Feed

from biblioteca.models import Libro

class UltimosLibrosFeed(Feed):
    title = "Feed libros publicados"
    link = "/libros/"
    description = "Ultimos libros publicados en la biblioteca digital."

    def items(self):
        return Libro.objects.order_by('-fecha_publicacion')[:5]

class UltimosLibrosAtom(UltimosLibrosFeed):
    feed_type = Atom1Feed
    subtitle = UltimosLibrosFeed.description
```

Y este es el URLconf asociada a cada entrada:

urls.py

```
from django.conf.urls import url
from biblioteca.feeds import UltimosLibrosFeed, UltimosLibrosAtom

urlpatterns = [
```

```
    # ...
    url(r'^feeds/$', UltimosLibrosFeed()),
    url(r'^atom/$', UltimosLibrosAtom()),
    # ...
]
```

El Framework Sitemap

Un *sitemap* es un fichero XML en tu sitio web que le indica a los indexadores de los motores de búsqueda cuan frecuentemente cambian tus páginas, así como la "importancia" relativa de ciertas páginas en relación con otras (siempre hablando de páginas de tu sitio Web). Esta información ayuda a los motores de búsqueda a indexar tu sitio.

Por ejemplo, esta es una parte del sitemap del sitio web de Django (http://www.djangoproject.com/sitemap.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url>
        <loc>http://www.djangoproject.com/documentation/</loc>
        <changefreq>weekly</changefreq>
        <priority>0.5</priority>
    </url>
    <url>
        <loc>http://www.djangoproject.com/documentation/0_90/</loc>
        <changefreq>never</changefreq>
        <priority>0.1</priority>
    </url>
...
</urlset>
```

Para más información sobre sitemaps, consulta http://www.sitemaps.org/.

El framework sitemap de Django automatiza la creación de este fichero XML si lo indicas expresamente en el código Python. Para crear un sitemap, debes simplemente escribir una clase Sitemap y hacer referencia a la misma en tu URLconf.

Instalación

Para instalar la aplicación sitemap, sigue los siguientes pasos:

1. Agrega 'django.contrib.sitemaps' a tu variable de configuración INSTALLED_APPS.

2. Asegúrate de que 'django.template.loaders.app_directories.Loader' está en tu variable de configuración TEMPLATE_LOADERS. Para que la aplicación sitemap encuentre las plantillas que necesita para funcionar.

3. Asegúrate de que tienes instalado el framework sites

La aplicación sitemap no instala tablas en la base de datos. La única razón de que esté en INSTALLED_APPS es que el cargador de plantillas Loader() pueda encontrar las plantillas incluidas.

Inicialización

Para activar la generación del sitemap en tu sitio Django, agrega la siguiente línea a tu URLconf:

```
from django.contrib.sitemaps.views import sitemap

urlpatterns =[
   url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
      name='django.contrib.sitemaps.views.sitemap')
]
```

Esta línea le dice a Django que construya un sitemap cuando un cliente accede a /sitemap.xml.

El nombre del fichero sitemap no es importante, pero la ubicación sí lo es. Los motores de búsqueda solamente indexan los enlaces en tu sitemap para el nivel de URL actual y anterior.

Por ejemplo, si sitemap.xml reside en tu directorio principal, el mismo puede hacer referencia a cualquier URL en tu sitio. Pero si tu sitemap reside en /content/sitemap.xml, solamente podrá hacer referencia a URLs que comiencen con /content/.

La vista sitemap toma un argumento extra: {'sitemaps': sitemaps}. sitemaps debe ser un diccionario que mapee una etiqueta corta de sección (p. ej. blog o consulta) a tu clase Sitemap (p.e., BlogSitemap o NewsSitemap).

También puede mapear una *instancia* de una clase Sitemap (p. ej. GenericSitemap(alguna_var)) en el mismo archivo urls.py.

Clases Sitemap

Una clase Sitemap es simplemente una clase Python que representa una "sección" de entradas en tu sitemap. Por ejemplo, una clase Sitemap puede representar todas las entradas de tu weblog, y otra puede representar todos los eventos de tu calendario.

En el caso más simple, todas estas secciones se unen en un único sitemap.xml, pero también es posible usar el framework para generar un índice sitemap que haga a referencia ficheros sitemap individuales, uno por sección (describiéndolo sintéticamente).

Las clases Sitemap debe ser una subclase de django.contrib.sitemaps.Sitemap. Estas pueden residir en cualquier parte del árbol de código.

Por ejemplo, asumamos que posees un sistema llamado biblioteca (si haz seguido los ejemplos ya tienes uno), con un modelo Autor, y quieres que tu sitemap incluya todos los enlaces a los autores.

Tu clase Sitemap debería verse así:

biblioteca/sitemap.py

```
from django.contrib.sitemaps import Sitemap
from biblioteca.models import Autor

class SitemapAutores(Sitemap):
   changefreq = "monthly"
   priority = 0.5

   def items(self):
```

```
        return Autor.objects.all()

    def lastmod(self, items):
        return items.ultimo_acceso
```

Y solo necesitas enlazar la clase SitemapAutores a la URLconf, así:

urls.py

```
from django.conf.urls import url
from django.contrib.sitemaps.views import sitemap

from biblioteca.sitemap import SitemapAutores

urlpatterns =[
    #...
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': {'sitemaps': SitemapAutores}}),
]
```

Declarar un Sitemap debería verse muy similar a declarar un Feed; esto es justamente un objetivo de diseño.

De forma similar a las clases Feed, los miembros de Sitemap pueden ser métodos o atributos. Consulta la sección "Un feed más complejo" para obtener más información sobre cómo funciona.

Una clase Sitemap puede definir los siguientes métodos/atributos:

- **items** (**requerido**): Provee una lista de objetos. Al framework no le importa qué tipo de objeto sean; todo lo que le importa es que los objetos sean pasados a los métodos location(), lastmod(), changefreq(), y priority().

- **location** (opcional): Provee la URL absoluta para el objeto dado. La "URL absoluta" significa una URL que no incluye el protocolo o el dominio.
  Estos son algunos ejemplos:

  ```
  Bien: '/foo/bar/'
  Mal: 'example.com/foo/bar/'
  Mal: 'http://example.com/foo/bar/'
  ```

  Si location no es provisto, el framework llamará al método get_absolute_url() en cada uno de los objetos retornados por items().

- **lastmod** (opcional): La fecha de "última modificación" del objeto, como un objeto datetime de Python.

- **changefreq** (opcional): Cuán a menudo el objeto cambia. Los valores posibles (según indican las especificaciones de Sitemaps) son:

  - 'always'
  - 'hourly'
  - 'daily'
  - 'weekly'
  - 'monthly'
  - 'yearly'
  - 'never'

- ***priority*** (opcional): Prioridad sugerida de indexado; entre 0.0 y 1.0. La prioridad por omisión de una página es 0.5; ver la documentación de http://sitemaps.org para más información de cómo funciona *priority.*

Accesos directos

El framework sitemap provee un conjunto de clases para los casos más comunes. Describiremos estos casos en las secciones a continuación.

FlatPageSitemap

La clase django.contrib.sitemaps.FlatPageSitemap apunta a todas las páginas planas definidas para el sitio actual y crea una entrada en el sitemap. Estas entradas incluyen solamente el atributo location – no lastmod, changefreq, o priority.

Para más información sobre Páginas Planas o "flatpages" consulta el *capítulo 14*.

GenericSitemap

La clase GenericSitemap (*Sitemap Genérico)* trabaja de forma bastante sencilla.

Para usarla, solo crea una instancia pasándola en una variable a sitemap en forma de diccionario. El único requerimiento es que el diccionario tenga una entrada a un queryset. También debe poseer una entrada un campo "date_field" que especifica un campo fecha para los objetos obtenidos del queryset. Esto será usado por el atributo lastmod en el sitemap generado. También puedes pasar argumentos por palabras clave (*keyword*) priority y changefreq al constructor GenericSitemap para especificar dichos atributos para todas las URLs.

Este es un ejemplo de URLconf parecido al anterior, solo que aquí estamos usando la clase genérica GenericSiteMap usando el mismo modelo Autor.

urls.py

```
from django.conf.urls import url
from django.contrib.sitemaps.views import sitemap
from django.contrib.sitemaps import GenericSitemap

from biblioteca.models import Autor

consulta = {
'queryset': Autor.objects.all(), # Un queryset
'date_field': 'ultimo_acceso',    #Un campo tipo fecha
}

sitemaps = {
'autores': GenericSitemap(consulta, priority=0.6, changefreq= 'always'),
}

urlpatterns =[
    #...
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps:}),
]
```

Crear un índice Sitemap

El framework sitemap también tiene la habilidad de crear índices sitemap que hagan referencia a ficheros sitemap individuales, uno por cada sección definida en tu diccionario sitemaps.

Las únicas diferencias de uso son:

- Usas dos vistas en tu URLconf: django.contrib.sitemaps.views.index y django.contrib.sitemaps.views.sitemap.
- La vista django.contrib.sitemaps.views.sitemap debe tomar un parámetro que corresponde a una palabra clave, llamado section. Por ejemplo:

```
urlpatterns =[
   #...
   url(r'^sitemap\.xml$','django.contrib.sitemaps.views.index',
      {'sitemaps': sitemaps}),
   url(r'^sitemap-(?P<section>.+).xml$', 'django.contrib.sitemaps.views.sitemap',
      {'sitemaps': sitemaps})
]
```

Así deberían verse las líneas relevantes en tu URLconf para el ejemplo anterior.

Esto genera automáticamente un fichero sitemap.xml que hace referencia a ambos ficheros sitemap-flatpages.xml y sitemap-autores.xml. La clase Sitemap y el diccionario sitemaps no cambian en absoluto.

Hacer ping a Google

Puedes desear hacer un "ping" a Google cuando tu sitemap cambia, para hacerle saber que debe reindexar tu sitio Web. El framework provee una función para hacer justamente eso:

```
django.contrib.sitemaps.ping_google().
```

■ **Nota:** ¡Primero regístrate con Google!

El comando ping_google() únicamente trabaja si haz registrado tu sitio con Google Webmaster Tools.

ping_google() toma un argumento opcional, sitemap_url, que debe ser la URL absoluta de tu sitemap (por ej., '/sitemap.xml'). Si este argumento no es provisto, ping_google() tratará de generar un sitemap realizando una búsqueda reversa en tu URLconf.

ping_google() lanza la excepción django.contrib.sitemaps.SitemapNotFound si no puede determinar la URL de tu sitemap.

Una forma útil de llamar a ping_google() es desde el método save():

```
from django.contrib.sitemaps import ping_google

class Libro(models.Model):
   # ...

   def save(self, force_insert=False, force_update=False):
```

```
        super(Libro, self).save(force_insert, force_update)
        try:
            ping_google()
        except Exception:
            #Por si ocurre una excepción
            Pass
```

Una solución más eficiente, sin embargo, sería llamar a ping_google() desde un script cron o un manejador de tareas. La función hace un pedido HTTP a los servidores de Google, por lo que no querrás introducir esa demora asociada a la actividad de red cada vez que se llame al método save().

Hacer ping a Google mediante manage.py

Una vez que la aplicacion sitemap es agregada a tu proyecto, puedes hacer ping a Google manualmente usando el comando ping_google mediante la línea de comandos así:

```
python manage.py ping_google [/sitemap.xml]
```

¿Qué sigue?

A continuación, seguiremos indagando más profundamente en las herramientas internas que Django nos ofrece. Él *capítulo 14* examina todas las herramientas que necesitas para proveer sitios personalizados: sesiones, usuarios, y autenticación.

CAPÍTULO 14

Sesiones, usuarios e inscripciones

Tenemos que confesar algo: hasta el momento hemos ignorado un aspecto absolutamente importante del desarrollo web. Hemos hecho la suposición de que el tráfico que visita nuestra web está compuesto por una masa amorfa de usuarios anónimos, que se precipitan contra nuestras cuidadosamente diseñadas páginas.

Esto no es verdad, claro. Los navegadores que consultan nuestras páginas tienen a personas reales detrás (la mayor parte del tiempo, al menos). Este es un hecho importantísimo y que no debemos ignorar: Lo mejor de Internet es que sirve para conectar personas, no máquinas. Si queremos desarrollar un sitio web realmente competitivo, antes o después tendremos que plantearnos como tratar a las personas que están detrás del navegador.

Por desgracia, no es tan fácil como podría parecer. El protocolo HTTP se diseñó específicamente para que fuera un protocolo *sin estado*, es decir, que cada petición y respuesta está totalmente aislada de las demás. No hay persistencia entre una petición y la siguiente, y ninguno de los atributos de la petición (dirección IP, identificador del agente, etc...) nos permite discriminar de forma segura y consistente las peticiones de una persona de las del resto.

En este capítulo aprenderemos como solucionar esta carencia de estados. Empezaremos al nivel más bajo (*cookies*), e iremos ascendiendo hasta las herramientas de alto nivel que nos permitirán gestionar sesiones, usuarios y altas o inscripciones de los mismos.

Cookies

Los desarrolladores de navegadores hace tiempo que se dieron cuenta de que esta carencia de estados iba a representar un problema para los desarrolladores web, y así fue como nacieron las *cookies* (literalmente *galleta*). Una cookie es una pequeña cantidad de información que el servidor delega en el navegador, de forma que este la almacena. Cada vez que el cliente web solicita una página del servidor, se le envía de vuelta la cookie.

Veamos con un poco más de detalle el funcionamiento. Cuando abrimos nuestro navegador y escribimos google.com, el navegador envía una solicitud HTTP a Google que empieza más o menos así:

```
GET / HTTP/1.1
Host: google.com
...
```

Cuando Google responde, la respuesta contiene algo parecido a esto:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie:
PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
            expires=Sun, 17-Jan-2038 19:14:07 GMT;
            path=/; domain=.google.com
Server: GWS/2.1
...
```

Fíjate en la línea que comienza con Set-Cookie. El navegador almacenará el valor indicado (PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671) y se lo volverá a enviar a Google cada vez que vuelva a acceder a alguna de sus páginas; de esa forma, la próxima vez que vuelvas a Google, la petición que enviará el navegador se parecerá a esta:

```
GET / HTTP/1.1
Host: google.com
Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671
...
```

Google puede saber ahora, gracias al valor de la *Cookie*, que eres la misma persona que accedió anteriormente. Este valor puede ser, por ejemplo, una clave en una tabla de la base de datos que almacene los datos del usuario. Con esa información, Google puede hacer aparecer tu nombre en la página (de hecho, lo hace).

Cómo definir y leer los valores de las cookies

A la hora de utilizar las capacidades de persistencia de Django, lo más probable es que uses las prestaciones de alto nivel para la gestión de sesiones y de usuarios, prestaciones que discutiremos un poco más adelante en este mismo capítulo. No obstante, ahora vamos a hacer una breve parada y veremos cómo leer y definir *cookies* a bajo nivel. Esto debería ayudarte a entender cómo funcionan el resto de las herramientas que veremos en el capítulo, y te será de utilidad si alguna vez tienes que trabajar con las cookies directamente.

Obtener los valores de las cookies que ya están definidas es muy fácil. Cada objeto de tipo petición, request, contiene un objeto COOKIES que se comporta como un diccionario; puedes usarlo para leer cualquier *cookie* que el navegador haya enviado a la vista:

```
def mostrar_color(request):
    if "color_favorito" in request.COOKIES:
        return HttpResponse("Tu color favorito es %s" % \
            request.COOKIES["color_favorito"])
    else:
        return HttpResponse("No tienes un color favorito.")
```

Definir los valores de las cookies es sólo un poco más complicado. Debes usar el método set_cookie() en un objeto de tipo HttpResponse. He aquí un ejemplo que define la *cookie* color_favorito utilizando el valor que se le pasa como parámetro GET:

```
def set_color(request):
    if "color_favorito" in request.GET:
        # Crea un objeto HttpResponse...
```

```
        response = HttpResponse("Tu color favorito es ahora %s" % \
            request.GET["color_favorito"])
        # ... y definimos la cookie en la respuesta
        response.set_cookie("color_favorito", request.GET["color_favorito"])
        return response
    else:
        return HttpResponse("No haz elegido un color favorito.")
```

Hay una serie de parámetros opcionales que puedes pasar a response.set_cookie() y que te permiten controlar determinadas características de la *cookie*, tal y como se muestra en la tabla 14-1.

Parámetro	Default	Descripción
max_age	None	El tiempo (en segundos) que la cookie debe permanecer activa. Si este parámetro es la *cookie*, desaparecerá automáticamente cuando se cierre el navegador.
expires	None	La fecha y hora en que la cookie debe expirar. Debe estar en el formato "Wdy, DD-Mth-YY HH:MM:SS GMT". Si se utiliza este parámetro, su valor tiene preferencia sobre el definido mediante max_age.
path	"/"	La ruta o *path* para la cual es válida la cookie. Los navegadores solo reenviarán la cookie a las páginas que estén en dicha ruta. Esto impide que se envíe esta cookie a otras secciones de la web. Es especialmente útil si no se tiene el control del nivel superior de directorios del servidor web.
domain	None	El dominio para el cual es válida la cookie. Se puede usar este parámetro para definir una cookie que sea apta para varios dominios. Por ejemplo, definiendo domain=".example.com" la cookie será enviada a los dominios www.example.com, www2.example.com y aun.otro.subdominio.example.com. Si a este parámetro no se le asigna ningún valor, la cookie solo será enviada al dominio que la definió.
secure	False	Si este valor se define como True, se le indica al navegador que sólo retorne esta cookie a las páginas que se accedan de forma segura (protocolo HTTPS en vez de HTTP).

Table 14-1. *Opciones de las Cookies*

Las cookies tienen doble filo

Puede que te hayas dado cuenta de algunos de los problemas potenciales que se presentan con esto de las cookies; vamos a ver algunos de los más importantes:

- El almacenamiento de los cookies es voluntario; los navegadores no dan ninguna garantía. De hecho, los navegadores permiten al usuario definir una política de aceptación o rechazo de las mismas. Para darte cuenta de lo muy usadas que son las cookies en la web actual, simplemente activa la

opción de "Avisar antes de aceptar cualquier cookie" y date un paseo por Internet.

A pesar de su uso habitual, las cookies son el ejemplo perfecto de algo que no es confiable. Esto significa que el desarrollador debe comprobar que el usuario está dispuesto a aceptar las cookies antes de confiar en ellas.

Aún más importante, *nunca* debes almacenar información fundamental en las cookies. La Web rebosa de historias de terror acerca de desarrolladores que guardaron información irrecuperable en las cookies del usuario, solo para encontrarse con que el navegador había borrado todos esos datos por cualquier razón.

- Las Cookies (especialmente aquellas que no se envían mediante HTTPS) no son seguras. Dado que los datos enviados viajan en texto claro, están expuestas a que terceras personas lean esa información, lo que se llama ataques de tipo *snooping* (por *snoop*, fisgonear, husmear). Por lo tanto, un atacante que tenga acceso al medio puede interceptar la cookie y leer su valor. El resultado de esto es que nunca se debe almacenar información confidencial en una cookie.

  Hay otro tipo de ataque, aún más insidioso, conocido como ataque *man-in-the-middle* o MitM (ataque de tipo Hombre-en-medio o Intermediario). Aquí, el atacante no solo intercepta la cookie, sino que además la usa para actuar ante el servidor como si fuera el usuario legítimo. Él *capítulo 19* describe en profundidad este tipo de ataques, así como formas de prevenirlo.

- Las Cookies ni siquiera son seguras para los servidores. La mayoría de los navegadores permiten manipular y editar de forma sencilla los contenidos de cookies individuales, y existen herramientas como mechanize (🌎 http://wwwsearch.sourceforge.net/mechanize/) que permiten a cualquiera que esté lo suficientemente motivado construir solicitudes HTTP a mano.

  Así que tampoco debemos almacenar en las cookies datos que sean fáciles de falsificar. El error habitual en este escenario consiste en almacenar algo así como IsLoggedIn=1 en una cookie cuando el usuario se ha validado. Te sorprendería saber cuántos sitios web cometen este tipo de error; no lleva más de unos segundos engañar a sus sistemas de "seguridad".

El entorno de sesiones de Django

Con todas estas limitaciones y agujeros potenciales de seguridad, es obvio que la gestión de las cookies y de las sesiones persistentes es el origen de muchos dolores de cabeza para los desarrolladores web. Por supuesto, uno de los objetivos de Django es evitar eficazmente estos dolores de cabeza, así que dispone de un entorno de sesiones diseñado para suavizar y facilitar todas estas cuestiones.

El entorno de sesiones te permite almacenar y recuperar cualquier dato que quieras basándote en la sesión del usuario. Almacena la información relevante solo en el servidor y abstrae todo el problema del envío y recepción de las cookies. Estas solo almacenan una versión codificada (*hash*) del identificador de la sesión, y ningún otro dato, lo cual te aísla de la mayoría de los problemas asociados con las cookies.

Veamos como activar las sesiones, y como usarlas en nuestras vistas.

Como activar las sesiones

Las sesiones se implementan mediante un poco de *middleware* y un modelo Django. Para activar las sesiones, necesitas seguir los siguientes pasos:

1. Editar el valor de MIDDLEWARE_CLASSES de forma que contenga 'django.contrib.sessions.middleware.SessionMiddleware'.

2. Comprueba que 'django.contrib.sessions' esté incluido en el valor de INSTALLED_APPS (y ejecuta el comando python manage.py migrate)

El archivo por defecto settings.py creado por el comando startproject activa estas dos características, así que a menos que las hayas borrado, es muy probable que no tengas que hacer nada para empezar a usar las sesiones.

Si lo que quieres en realidad es no usar sesiones, deberías quitar la referencia a SessionMiddleware de MIDDLEWARE_CLASSES y borra 'django.contrib.sessions' de INSTALLED_APPS. Esto te ahorrará sólo un poco de sobrecarga, pero toda ayuda es buena.

Usando las sesiones en las vistas

Cuando están activadas las sesiones, los objetos HttpRequest –el primer argumento de cualquier función que actúe como una vista en Django tendrán un atributo llamado session, que se comporta igual que un diccionario. Se puede leer y escribir en él de la misma forma en que lo harías con un diccionario normal. Por ejemplo, podrías usar algo como esto en una de tus vistas:

```
# Asigna un valor a la sesión:
request.session["fav_color"] = "blue"
# Trae el valor de la sesión – puede ser llamado en diferentes vistas o e muchas
# peticiones  t(o en  ambas):
fav_color = request.session["fav_color"]

# Limpia el item para la sesion:
del request.session["fav_color"]

# Verifica que la sesión contenga una clave:
if "fav_color" in request.session:
...
```

También puedes usar otros métodos propios de un diccionario como keys() o items() en request.session.

Hay dos o tres reglas muy sencillas para usar eficazmente las sesiones en Django:

- Debes usar sólo cadenas de texto normales como valores de clave en request.session, en vez de, por ejemplo, enteros, objetos, etc. Esto es más un convenio que un regla en el sentido estricto, pero merece la pena seguirla.

- Los valores de las claves de una sesión que empiecen con el carácter subrayado están reservados para uso interno de Django. En la práctica, sólo hay unas pocas variables así, pero, a no ser que sepas lo que estás haciendo (y estés dispuesto a mantenerte al día en los cambios internos de Django), lo mejor que puedes hacer es evitar usar el carácter subrayado como prefijo en tus propias variables; eso impedirá que Django pueda interferir con tu aplicación.

- Nunca reemplaces request.session por otro objeto, y nunca accedas o modifiques sus atributos. Utilízalo sólo como si fuera un diccionario.

Veamos un ejemplo rápido. Esta vista simplificada define una variable "ya_comento" como True después de que el usuario haya publicado un comentario. Es una forma sencilla (aunque no particularmente segura) de impedir que el usuario publique dos veces el mismo comentario:

```
def postear_comentario(request):
    if request.method != 'POST':
        raise Http404('Unicamente se permiten subir POSTs')

    if 'comentario' not in request.POST:
        raise Http404('Comentario no enviado')

    if request.session.get('ya_comento', False):
       return HttpResponse("Ya haz comentado")

    c = comments.Comment(comment=request.POST['comment'])
    c.save()
    request.session[' ya_comento'] = True

    return HttpResponse('¡Gracias por comentar!')
```

Esta vista simplificada permite que un usuario se identifique como tal en nuestras páginas:

```
def login(request):
    if request.method != 'POST':
        raise Http404('Únicamente se permiten métodos POSTs')
    try:
        m = Member.objects.get(username=request.POST['username'])
        if m.password == request.POST['password']:
            request.session['member_id'] = m.id
            return HttpResponseRedirect('/you-are-logged-in/')
    except Member.DoesNotExist:
        return HttpResponse("Tu nombre de usuario y contraseña no coinciden.")
```

Y esta le permite cerrar o salir de la sesión:

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
       pass
    return HttpResponse("Haz salido de la sesión")
```

■ **Nota:** En la práctica, esta sería una forma pésima de validar a tus usuarios. El mecanismo de autentificación que presentaremos un poco más adelante realiza esta tarea de forma mucho más segura y robusta. Los ejemplos son deliberadamente simples para que se comprendan con más facilidad.

Comprobar las configuraciones de las cookies

Como ya mencionamos, no se puede confiar en que cualquier navegador sea capaz de aceptar *cookies*. Por ello, Django incluye una forma fácil de comprobar que el cliente del usuario disponga de esta capacidad. Sólo es necesario llamar a la función request.session.set_test_cookie() en una vista, y comprobar posteriormente, en otra vista distinta, el resultado de llamar a request.session.test_cookie_worked().

Esta división un tanto extraña entre las llamadas a set_test_cookie() y test_cookie_worked() se debe a la forma es que trabajan las *cookies*. Cuando se define una *cookie*, no tienes forma de saber si el navegador la ha aceptado realmente hasta la siguiente solicitud.

Es una práctica recomendable llamar a la función delete_test_cookie() para limpiar la cookie de prueba después de haberla usado. Lo mejor es hacerlo justo después de haber verificado que las *cookies* funcionan.

He aquí un ejemplo típico de uso:

```
def login(request):
    # Si  la petición es POST...
    if request.method == 'POST':
        # Prueba que la cookie trabaje (ver abajo):
        if request.session.test_cookie_worked():
            # Si la cookie trabaja, se borra.
            request.session.delete_test_cookie()
                # En la práctica necesitas alguna lógica para verificar el nombre de
                # usuario/contraseña...
                return HttpResponse("Tu estas autentificado.")
        # Si la prueba falla, se muestra un mensaje de error. En un sitio real deberías
        # mostrar un mensaje amigable.
        else:
            return HttpResponse("Por favor habilita las cookies otra vez.")
    # Si no es POST, envía la cookie con el formulario de autentificación
    request.session.set_test_cookie()
    return render(request, 'foo/login_form.html')
```

■ **Nota:** De nuevo, las funciones de autentificación ya definidas en el entorno se encargan de realizar estos chequeos por ti.

Usar sesiones fuera de las vistas

Internamente, cada sesión es simplemente un modelo de entidad de Django como cualquier otro, definido en django.contrib.sessions.models. Cada sesión se identifica gracias a un *hash* pseudo-aleatorio de 32 caracteres, que es el valor que se almacena

en la cookie. Dado que es un modelo normal, puedes acceder a las propiedades de las sesiones usando la API de acceso a la base de datos de Django:

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 14)
```

Para poder acceder a los datos de la sesión, hay que usar el método get_decoded(). Esto se debe a que estos datos, que consistían en un diccionario, están almacenados codificados:

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

Cuándo se guardan las sesiones

Django, en principio, solo almacena la sesión en la base de datos si ésta ha sido modificada; es decir, si cualquiera de los valores almacenados en el diccionario es asignado o borrado. Esto puede dar lugar a algunos errores sutiles, como se indica en el último ejemplo:

```
# La sesión ha sido modificada.
request.session['foo'] = 'bar'

# La sesión ha sido modificada.
del request.session['foo']

# La sesión ha sido modificada.
request.session['foo'] = {}

# La sesión NO ha sido modificada, porque se altero
# request.session['foo'] en lugar de  request.session.
request.session['foo']['bar'] = 'baz'
```

Se puede cambiar este comportamiento, especificando la opción SESSION_SAVE_EVERY_REQUEST a True. Si lo hacemos así, Django almacenará la sesión en la base de datos en cada petición, incluso si no se ha modificado ninguno de sus valores.

Fíjate que la cookie de sesión sólo se envía cuando se ha creado o modificado una sesión. Si SESSION_SAVE_EVERY_REQUEST está como True, la cookie de sesión será reenviada en cada petición. De forma similar, la sección de expiración (''expires'') se actualizará cada vez que se reenvíe la cookie.

Sesiones breves frente a sesiones persistentes

Es posible que te hayas fijado en que la cookie que nos envió Google al principio del capítulo contenía el siguiente texto expires=Sun, 17-Jan-2038 19:14:07 GMT;. Las Cookies pueden incluir opcionalmente una fecha de expiración, que informa al navegador el momento en que se debe desechar por inválida. Si la cookie no contiene

ningún valor de expiración, el navegador entiende que esta debe expirar en el momento en que se cierra el propio navegador. Se puede controlar el comportamiento del entorno para que use cookies de este tipo, breves, ajustando en valor de la opción SESSION_EXPIRE_AT_BROWSER_CLOSE.

El valor por omisión de la opción SESSION_EXPIRE_AT_BROWSER_CLOSE es False, lo que significa que las cookies serán almacenadas en el navegador del usuario durante SESSION_COOKIE_AGE segundos (cuyo valor por defecto es de dos semanas, o 1.209.600 segundos). Estos valores son adecuados si no quieres obligar a tus usuarios a validarse cada vez que abran el navegador y accedan a tu página.

Si SESSION_EXPIRE_AT_BROWSER_CLOSE se establece a True, Django usará cookies que se invalidarán cuando el usuario cierre el navegador.

Otras características de las sesiones

Además de las características ya mencionadas, hay otros valores de configuración que influyen en la gestión de sesiones con Django, tal y como se muestra en la tabla 14-2.

Opción	Descripción	Default
SESSION_COOKIE_DOMAIN	El Dominio a utilizar por la cookie de sesión. Se puede utilizar, por ejemplo, el valor ".lawrence.com" para utilizar la cookie en diferentes subdominios. El valor None indica una cookie estándar.	None
SESSION_COOKIE_NAME	El nombre de la cookie de sesiones. Puede ser cualquier cadena de texto.	"sessionid"
SESSION_COOKIE_SECURE	Indica si se debe usar una cookie segura para la cookie de sesión. Si el valor es True, la cookie se marcará como segura, lo que significa que sólo se podrá utilizar mediante el protocolo HTTPS.	False

Tabla 14-2. *Valores de configuración que influyen en el comportamiento de las cookies*

DETALLES TÉCNICOS

Para los más curiosos, he aquí una serie de notas técnicas acerca de algunos aspectos interesantes de la gestión interna de las sesiones:

El diccionario de la sesión acepta cualquier objeto Python capaz de ser serializado con pickle. Véase la documentación del módulo pickle incluido en la biblioteca estándar de Python para más información.

Los datos de la sesión se almacenan en una tabla en la base de datos llamada django_session.

Los datos de la sesión son suministrados bajo demanda. Si nunca accedes al atributo request.session, Django nunca accederá a la base de datos.

Django sólo envía la cookie si tiene que hacerlo. Si no modificas ningún valor de la sesión, no reenvía la cookie (a no ser que hayas definido SESSION_SAVE_EVERY_REQUEST como True).

El entorno de sesiones de Django se basa entera y exclusivamente en las cookies. No almacena la información de la sesión en las URL, como recurso extremo en el caso de que no se puedan utilizar las cookies, como hacen otros entornos (PHP, JSP).

Esta es una decisión tomada de forma consciente. Poner los identificadores de sesión en las URL no solo hace que las direcciones sean más feas, también hace que el sistema sea vulnerable ante un tipo de ataque en que se roba el identificador de la sesión utilizando la cabecera Referer.

Si aun te pica la curiosidad, el código fuente es bastante directo y claro, mira en django.contrib.sessions para más detalles.

Usuarios e identificación

Estamos ya a medio camino de poder conectar los navegadores con la Gente de Verdad™. Las sesiones nos permiten almacenar información a lo largo de las diferentes peticiones del navegador; la segunda parte de la ecuación es utilizar esas sesiones para validar al usuario, es decir, permitirle hacer *login*. Por supuesto, no podemos simplemente confiar en que los usuarios sean quien dice ser, necesitamos autentificarlos de alguna manera.

Naturalmente, Django nos proporciona las herramientas necesarias para tratar con este problema tan habitual (y con muchos otros). El sistema de autentificación de usuarios de Django maneja cuentas de usuarios, grupos, permisos y sesiones basadas en cookies. El sistema también es llamado sistema *aut/aut* (autenticaficación y autorización). El nombre implica que, a menudo, tratar con los usuarios implica dos procesos. Se necesita:

- Verificar (*autentificación*) que un usuario es quien dice ser (Normalmente comprobando un nombre de usuario y una contraseña contra una tabla de una base de datos)

- Verificar que el usuario está autorizado (*autorización*) a realizar una operación determinada (normalmente comprobando una tabla de permisos)

Siguiendo estos requerimientos, el sistema aut/aut de Django consta de los siguientes componentes:

1. **Usuarios**: Personas registradas en tu sitio web

2. **Permisos**: Valores binarios (Si/No) que indican si un usuario puede o no realizar una tarea determinada.

3. **grupos:** Una forma genérica de aplicar etiquetas y permisos a más de un usuario.

4. **mensajes:** Un mecanismo sencillo que permite enviar y mostrar mensajes del sistema usando una cola.

5. ***Perfiles***: Un mecanismo que permite extender los objetos de tipo usuario con campos adicionales.

Si ya has utilizado la herramienta de administración (descrita en el *capítulo 6*), habrás visto muchas de estas utilidades, y si has modificado usuarios y grupos con dicha herramienta, ya has modificado las tablas en las que se basa el sistema aut/aut.

Habilitar el soporte para autentificación

Al igual que ocurría con las sesiones, el sistema de autentificación viene incluido como una aplicación en el módulo django.contrib, y necesita ser instalado. De igual manera, viene instalado por defecto, por lo que solo es necesario seguir los siguientes pasos si previamente la has desinstalado:

- Comprueba que el sistema de sesiones esté activo, tal y como se explico previamente en este capítulo. Seguir la pista de los usuarios implica usar cookies, y por lo tanto necesitamos el entorno de sesiones operativo.

- Incluye 'django.contrib.auth' dentro de tu INSTALLED_APPS y ejecuta los comandos makemigration y migrate.
- Asegúrate de que 'django.contrib.auth.middleware.AuthenticationMiddleware' está incluido en MIDDLEWARE_CLASSES *después de* SessionMiddleware.

Una vez resuelto este tema, ya estamos preparados para empezar a lidiar con los usuarios en nuestras vistas. La principal interfaz que usarás para trabajar con los datos del usuario dentro de una vista es request.user; es un objeto que representa al usuario que está conectado en ese momento. Si no hay ningún usuario conectado, este objeto será una instancia de la clase AnonymousUser (veremos más sobre esta clase un poco más adelante).

Puedes saber fácilmente si el usuario está identificado o no con el método is_authenticated():

```
if request.user.is_authenticated():
   # Hacer algo con usuarios autentificados.
else:
   # Hacer algo con usuarios autentificados.
```

Utilizando usuarios

Una vez que ya tienes un usuario (normalmente mediante request.user, aunque también puede ser por otros métodos, que se describirán en breve) dispondrás de una serie de campos de datos y métodos asociados al mismo. Los objetos de la clase AnonymousUser emulan *parte* de esta interfaz, pero no toda, por lo que es preferible comprobar el resultado de user.is_authenticated() antes de asumir de buena fe que nos encontramos ante un usuario legítimo. Las tablas 14-3 y 14-4 listan todos los campos y métodos, respectivamente, de los objetos de la clase User.

Campos de los objetos User

Campo	Descripción
username	Obligatorio; 30 caracteres como máximo. Sólo acepta caracteres alfanuméricos (letras, dígitos y el carácter subrayado).
first_name	Opcional; 30 caracteres como máximo.
last_name	Opcional; 30 caracteres como máximo.
email	Opcional. Dirección de correo electrónico.
password	Obligatorio. Un código de comprobación (*hash*), junto con otros metadatos de la contraseña. Django nunca almacena la contraseña en crudo. Véase la sección "Cambia contraseñas" para más información
is_staff	Booleano. Indica que el usuario puede acceder a las secciones de administración.
is_active	Booleano. Indica que la cuenta puede ser usada para identificarse. Se puede poner a False para deshabilitar a un usuario sin tener que borrarlo de la tabla.
is_superuser	Booleano. Señala que el usuario tiene todos los permisos, aún cuando no se le hayan asignado explícitamente
last_login	Fecha y hora de la última vez que el usuario se identificó. Se asigna automáticamente a la fecha actual por defecto.
date_joined	Fecha y hora en que fue creada esta cuenta de usuario. Se asigna automáticamente a la fecha actual en su momento.

Tabla 14-3 *Campos de los objetos User*

Métodos de los objetos User

Método	Descripción
is_authenticated()	Siempre devuelve True para usuario reales. Es una forma de determinar si el usuario se ha identificado. esto no implica que posea ningún permiso, y tampoco comprueba que la cuenta esté activa. Sólo indica que el usuario se ha identificado con éxito.
is_anonymous()	Devuelve True sólo para usuarios anónimos, y False para usuarios "reales". En general, es preferible usar el método is_authenticated().
get_full_name()	Devuelve la concatenación de los campos first_name y last_name, con un espacio en medio.
set_password(passwd)	Cambia la contraseña del usuario a la cadena de texto en claro indicada, realizando internamente las operaciones necesarias para calcular el código de comprobación o *hash* necesario. Este método *no* guarda el objeto User.
check_password(passwd)	devuelve True si la cadena de texto en claro que se le pasa coincide con la contraseña del usuario. Realiza internamente las operaciones necesarias

	para calcular los códigos de comprobación o *hash* necesarios.
get_group_permissions()	Devuelve una lista con los permisos que tiene un usuario, obtenidos a través del grupo o grupos a las que pertenezca.
get_all_permissions()	Devuelve una lista con los permisos que tiene concedidos un usuario, ya sea a través de los grupos a los que pertenece o bien asignados directamente.
has_perm(perm)	Devuelve True si el usuario tiene el permiso indicado. El valor de perm está en el formato `"package.codename". Si el usuario no está activo, siempre devolverá False.
has_perms(perm_list)	Devuelve True si el usuario tiene *todos* los permisos indicados. Si el usuario no está activo, siempre devolverá False.
has_module_perms(app_label)	Devuelve True si el usuario tiene algún permiso en la etiqueta de aplicación indicada, app_label. Si el usuario no está activo, siempre devolverá False.
get_and_delete_messages()	Devuelve una lista de mensajes (objetos de la clase Message) de la cola del usuario, y los borra posteriormente.
email_user(subj, msg)	Envía un correo electrónico al usuario. El mensaje aparece como enviado desde la dirección indicada en el valor DEFAULT_FROM_EMAIL. Se le puede pasar un tercer parámetro opcional, from_email, para indicar otra dirección de remite distinta.

Tabla 14-4. Métodos de los objetos User

Por último, los objetos de tipo User mantienen dos campos de relaciones múltiples o muchos-a-muchos: Grupos y permisos (groups y permissions). Se puede acceder a estos objetos relacionados de la misma manera en que se usan otros campos múltiples:

```
# Asignar un usuario a un grupo:
>>>myuser.groups = group_list
# Agregar un usuario  a un grupo:
>>>myuser.groups.add(group1, group2,...)
# Quitar un usuario de algún grupo:
>>>myuser.groups.remove(group1, group2,...)
#Quitar un usuario de todos los grupos:
>>>myuser.groups.clear()
#Los permisos trabajan de la misma forma
>>>myuser.permissions = [permission_list]
>>>myuser.permissions.add(permission1, permission2, ...)
>>>myuser.permissions.remove(permission1, permission2, ...)
>>>myuser.permissions.clear()
```

Iniciar y cerrar sesión

Django proporciona vistas predefinidas para gestionar la entrada del usuario, (el momento en que se identifica), y la salida, (es decir, cuando cierra la sesión), además de otros trucos ingeniosos. Pero antes de entrar en detalles, veremos cómo hacer que el usuario puedan iniciar y cerrar la sesión "a mano". Django incluye dos funciones para realizar estas acciones, en el módulo django.contrib.auth: authenticate() y login().

Para autentificar un identificador de usuario y una contraseña, se utiliza la función authenticate(). esta función acepta dos parámetros , username y password, y devuelve un objeto de tipo User si la contraseña es correcta para el identificador de usuario. Si falla la comprobación (ya sea porque sea incorrecta la contraseña o porque sea incorrecta la identificación del usuario), la función devolverá None:

```
>>> from django.contrib import auth
>>> user = auth.authenticate(username='john', password='secret')
>>> if user is not None:
...       print ("¡Correcto!")
...    else:
...       print ("¡Oops, algo anda mal!")
```

La llamada a authenticate() sólo verifica las credenciales del usuario. Todavía hay que realizar una llamada a login() para completar el inicio de sesión. La llamada a login() acepta un objeto de la clase HttpRequest y un objeto User y almacena el identificador del usuario en la sesión, usando el entorno de sesiones de Django.

El siguiente ejemplo muestra el uso de ambas funciones, authenticate() y login(), dentro de una vista:

```
from django.contrib import auth

def vista_login(request):
   username = request.POST.get('username', '')
   password = request.POST.get('password', '')
   user = auth.authenticate(username=username, password=password)
   if user is not None and user.is_active:
      # Contraseña correcta y usuario marcado como "activo"
      auth.login(request, user)

      # Redireccciona a una página de entrada correcta.
      return HttpResponseRedirect("/account/loggedin/")
   else:
      # Muestra una página de error
      return HttpResponseRedirect("/account/invalid/")
```

Para cerrar la sesión, se puede llamar a django.contrib.auth.logout() dentro de una vista.

Necesita que se le pase como parámetro un objeto de tipo HttpRequest, y no devuelve ningún valor:

```
from django.contrib import auth

def logout(request):
   auth.logout(request)
   # Redireccciona a una página de entrada correcta.
   return HttpResponseRedirect("/account/loggedout/")
```

La llamada a logout() no produce ningún error, aun si no hubiera ningún usuario conectado.

En la práctica, no es normalmente necesario escribir tus propias funciones para realizar estas tareas; el sistema de autentificación viene con un conjunto de vistas predefinidas para ello. El primer paso para utilizar las vistas de autentificación es mapearlas en tu URLconf.

Necesitas modificar tu código hasta tener algo parecido a esto:

```
from django.contrib.auth.views import login, logout

urlpatterns = [
    # mas patrones aqui...
   url(r'^accounts/login/$', login),
   url(r'^accounts/logout/$', logout),
]
```

/accounts/login/ y */accounts/logout/* son las URL por defecto que usa Django para estas vistas.

Por defecto, la vista de login utiliza la plantilla definida en registration/login.html (puedes cambiar el nombre de la plantilla utilizando un parámetro opcional, template_name). El formulario necesita contener un campo llamado username y otro llamado password. Una plantilla de ejemplo podría ser esta:

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

{% block content %}

{% if form.errors %}
    <p class="error">Lo sentimos, la contraseña y el nombre de usuario no son
        validos</p>
{% endif %}

<form action="" method="post">
    <label for="username">User name:</label>
    <input type="text" name="username" value="" id="username">
    <label for="password">Password:</label>
    <input type="password" name="password" value="" id="password">
    <input type="submit" value="login" />
    <input type="hidden" name="next" value="{{ next|escape }}" />
 </form>
{% endblock %}
```

Si el usuario se identifica correctamente, su navegador será redirigido a /accounts/profile/. Puedes indicar una dirección distinta especificando un tercer campo (normalmente oculto) que se llame next, cuyo valor debe ser la URL a redireccionar después de la identificación. También puedes pasar este valor como un parámetro GET a la vista de identificación y se añadirá automáticamente su valor al contexto en una variable llamada next, que puedes incluir ahora en un campo oculto.

La vista de cierre de sesión se comporta de forma un poco diferente. Por defecto utiliza la plantilla definida en registration/logged_out.html (que normalmente contiene un mensaje del tipo “Ha cerrado su sesión”). No obstante, se puede llamar a esta vista con un parámetro extra, llamado next_page, que indicaría la vista a la que se debe redirigir una vez efectuado el cierre de la sesión.

Limitar el acceso a los usuarios identificados

Por supuesto, la razón de haber implementado todo este sistema es permitirnos limitar el acceso a determinadas partes de nuestro sitio.

La forma más simple y directa de limitar este acceso es comprobar el resultado de llamar a la función request.user.is_authenticated() y redirigir a una página de identificación, si procede:

```
from django.shortcuts import redirect

def mi_vista(request):
    if not request.user.is_authenticated():
        return redirect('/login/?next=%s' % request.path)
    # ...
```

O quizás mostrar un mensaje de error:

```
from django.shortcuts import render

def mi_vista(request):
    if not request.user.is_authenticated():
        return render(request, 'myapp/login_error.html')
    # ...
```

Si se desea abreviar, se puede usar el decorador **login_required** sobre las vistas que nos interese proteger:

```
from django.contrib.auth.decorators import login_required

@login_required
def mi_vista(request):
    # ...
```

Esto es lo que hace el decorador: " login_required"

- Si el usuario no está identificado, redirige a la dirección /accounts/login/, incluyendo la url actual como un parámetro con el nombre next, por ejemplo /accounts/login/?next=/polls/3/.

- Si el usuario está identificado, ejecuta la vista sin ningún cambio. La vista puede asumir sin problemas que el usuario está identificado correctamente.

Limitar el acceso a usuarios que pasan una prueba

Se puede limitar el acceso basándose en ciertos permisos o en algún otro tipo de prueba, o proporcionar una página de identificación distinta de la vista por defecto, y las dos cosas se hacen de manera similar.

La forma más cruda es ejecutar las pruebas que queremos hacer directamente en el código de la vista. Por ejemplo, para comprobar que el usuario está identificado y que, además, tenga asignado el permiso por ejemplo: biblioteca.votar (se explicará esto de los permisos con más detalle dentro de poco) haríamos:

```
def votar(request):
    if request.user.is_authenticated() and request.user.has_perm('biblioteca.votar')):
        # Votar aqui:
    else:
        return HttpResponse("Puedes  votar.")
```

De nuevo, Django proporciona una forma abreviada llamada user_passes_test(). Requiere que se la pasen unos argumentos y genera un decorador especializado para cada situación en particular:

```
def puede_votar(user):
    return user.is_authenticated() and user.has_perm("biblioteca.puede_votar")

@user_passes_test(puede_votar, login_url="/login/")
def votar(request):
    ...
```

El decorador user_passes_test tiene un parámetro obligatorio: un objeto que se pueda llamar (normalmente una función) y que a su vez acepte como parámetro un objeto del tipo User, y devuelva True si el usuario puede acceder y False en caso contrario. Es importante destacar que user_passes_test no comprueba automáticamente que el usuario esté identificado; esa es una comprobación que se debe hacer explícitamente.

En este ejemplo, hemos usado también un segundo parámetro opcional, login_url, que te permite indicar la url de la página que el usuario debe utilizar para identificarse (/accounts/login/ por defecto).

Comprobar si un usuario posee un determinado permiso es una tarea muy frecuente, así que Django proporciona una forma abreviada para estos casos: El decorador permission_required(). Usando este decorador, el ejemplo anterior se podría codificar así:

```
from django.contrib.auth.decorators import permission_required

@permission_required('biblioteca.puede_votar', login_url="/login/")
def votar(request):
    # ...
```

El decorador permission_required() también acepta el parámetro opcional login_url, de nuevo con el valor /accounts/login/ en caso de omisión.

Limitar el acceso a vistas genéricas

Una de las preguntas más frecuentes en la lista de usuarios de Django trata de cómo limitar el acceso a una vista genérica. Para conseguirlo, tienes que usar un recubrimiento sencillo alrededor de la vista que quieres proteger o apuntar en tu URLconf al recubrimiento en vez de a la vista genérica.

Puedes cambiar el decorador login_required por cualquier otro que quieras usar, como es lógico.

La forma más simple de limitar el acceso a una vista genérica basada en clases, es poner un decorador en la URLconf, en el método as_view() así:

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView
```

```
from .views import VoteView

urlpatterns = [
    #...
    url(r'^about/', login_required(TemplateView.as_view(
        template_name="secreto.html"))),
    url(r'^votar/', permission_required('biblioteca.puede_votar')(VistaVotar.as_view())),
]
```

Para limitar el acceso a una vista basada en una clase, solo decora el método dispatch() así:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

class VistaProtegida(TemplateView):
    template_name = 'secreto.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(VistaProtegida, self).dispatch(*args, **kwargs)
```

Gestionar usuarios, permisos y grupos

La forma más fácil de gestionar el sistema de autentificación es a través de la interfaz de administración admin. Él *capítulo 6* describe cómo usar esta interfaz para modificar los datos de los usuarios y controlar sus permisos y accesos, y la mayor parte del tiempo esa es la forma más adecuada de gestión.

A veces, no obstante, hace falta un mayor control, y para eso podemos utilizar las llamadas a bajo nivel que describiremos en este capítulo.

Crear usuarios

Puedes crear usuarios con el método create_user():

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user(username='john',
...                                 email='jlennon@beatles.com',
...                                 password='glass onion')
>>>
#En este punto, user es una instancia de la clase User, preparada para
# ser almacenada en la base de datos (create_user() no llama al método save()). Este
# te permite cambiar algunos de sus atributos antes de guardarlos, si quieres:
>>> user.last_name = 'Lennon'
>>> user.is_staff = True
>>> user.save()
```

Puedes usar django-admin para realizar este trabajo interactivamente

Cambia contraseñas

Puedes cambiar las contraseña de un usuario llamando a set_password():

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(username='john')
>>> user.set_password('nueva-contraseña')
>>> user.save()
```

No debes modificar directamente el atributo password, a no ser que tengas muy claro lo que estás haciendo. La contraseña se almacena en la base de datos en forma de código de comprobación (*salted hash*) y, por tanto, debe ser modificada sólo a través de este método.

Para ser más exactos, el atributo password de un objeto User es una cadena de texto con el siguiente formato:

```
hashtype$salt$hash
```

Es decir, el tipo de hash, el grano de sal (*salt*) y el código hash propiamente dicho, separados entre sí por el carácter dólar ($).

El valor de hashtype puede ser sha1 (por defecto) o md5, el algoritmo usado para realizar una transformación *hash* de un solo sentido sobre la contraseña. El grano de sal es una cadena de texto aleatoria que se utiliza para aumentar la resistencia de esta codificación frente a un ataque por diccionario. Por ejemplo:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

Las funciones User.set_password() y User.check_password() manejan todos estos detalles y comprobaciones de forma transparente.

¿Tengo que echar sal a mi ordenador?

No, la sal de la que hablamos no tiene nada que ver con ninguna receta de cocina; es una forma habitual de aumentar la seguridad a la hora de almacenar una contraseña.

Una función *hash* es una función criptográfica, que se caracteriza por ser de un solo sentido; es decir, es fácil calcular el código *hash* de un determinado valor, pero es prácticamente imposible reconstruir el valor original partiendo únicamente del código hash.

Si almacenáramos las contraseñas como texto en claro, cualquiera que pudiera obtener acceso a la base de datos podría saber sin ninguna dificultad todas las contraseñas al instante. Al guardar las contraseñas en forma de códigos *hash* se reduce el peligro en caso de que se comprometa la seguridad de la base de datos.

No obstante, un atacante que pudiera acceder a la base de datos podría ahora realizar un ataque por fuerza bruta, calculando los códigos *hash* de millones de contraseñas distintas y comparando esos códigos con los que están almacenados en la base de datos. Este llevará algo de tiempo, pero menos de lo que parece, los ordenadores son increíblemente rápidos.

Para empeorar las cosas, hay disponibles públicamente lo que se conoce como tablas arco iris (*rainbow tables*), que consisten en valores *hash* precalculados de millones de contraseñas de uso habitual. Usando una tabla arco iris, un atacante puede romper la mayoría de las contraseñas en segundos.

Para aumentar la seguridad, se añade un valor inicial aleatorio y diferente a cada contraseña antes de obtener el código *hash*. Este valor aleatorio es el "grano de sal".

Como cada grano de sal es diferente para cada password se evita el uso de tablas arco iris, lo que obliga al atacante a volver al sistema de ataque por fuerza bruta, que

a su vez es más complicado al haber aumentado la entropía con el grano de sal. Otra ventaja es que si dos usuarios eligen la misma contraseña, al añadir el grano de sal los códigos hash resultantes serán diferentes.

Aunque esta técnica no es, en términos absolutos, la más segura posible, ofrece un buen compromiso entre seguridad y conveniencia.

El alta de usuarios

Podemos usar estas herramientas de bajo nivel para crear vistas que permitan al usuario darse de alta. Prácticamente todos los desarrolladores quieren implementar el alta del usuario a su manera, por lo que Django te da la opción de crearte tu propia vista para ello. Afortunadamente, es muy fácil de hacer.

La forma más sencilla es escribir una pequeña vista que pregunte al usuario los datos que necesita y con ellos se cree directamente el usuario. Django proporciona un formulario prefabricado que se puede usar con este fin, como se muestra en el siguiente ejemplo:

biblioteca/forms.py

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.http import HttpResponseRedirect
from django.shortcuts import render

def registrar(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            new_user = form.save()
            return HttpResponseRedirect("/libros/")
    else:
        form = UserCreationForm()
    return render(request, "biblioteca/registro.html", {
        'form': form,
    })
```

Este formulario asume que existe una plantilla llamada biblioteca/registro.html. Esta plantilla podría consistir en algo parecido a esto:

templates/biblioteca/registro.html

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

{% block title %}Crea una cuenta{% endblock %}

{% block content %}
  <h1> Crea una cuenta de usuario</h1>

  <form action="" method="post"> {% csrf_token %}
      {{ form.as_p }}
      <input type="submit" value="Crea una cuenta">
  </form>
{% endblock %}
```

Usar información de autentificación en plantillas

El usuario actual, así como sus permisos, están disponibles en el contexto de la plantilla cuando usas RequestContext (véase *Capítulo 10*).

■Nota: Técnicamente hablando, estas variables están disponibles en el contexto de la plantilla sólo si usas RequestContext *y* en la configuración está incluido el valor "django.core.context_processors.auth" en la opción TEMPLATE_CONTEXT_PROCESSORS, que es el valor que viene predefinido cuando se crea un proyecto. Como ya se comentó, véase él *capítulo 10* para más información.

Cuando se usa RequestContext, el usuario actual (ya sea una instancia de User o de AnonymousUser) es accesible en la plantilla con el nombre {{ user }}:

```
{% if user.is_authenticated %}
  <p>Bienvenido, {{ user.username }}. Gracias por identificarte.</p>
{% else %}
  <p>Bienvenido, nuevo usuario. Por favor identifícate.</p>
{% endif %}
```

Los permisos del usuario se almacenan en la variable {{ perms }}. En realidad, es una forma simplificada de acceder a un par de métodos sobre los permisos que veremos en breve.

Hay dos formas de usar este objeto perms. Puedes usar {{ perms.polls }} para comprobar si un usuario tienen *algún* permiso para una determinada aplicación, o se puede usar una forma más específica, como {{ perms.polls.can_vote }}, para comprobar si el usuario tiene concedido un permiso en concreto.

Por lo tanto, se pueden usar estas comprobaciones en sentencias {% if %}:

```
{% if perms.biblioteca %}
  <p>Tienes permisos para  publicar libros.</p>
  {% if perms.biblioteca.puede_votar %}
    <p>¡Puedes votar en las encuestas!</p>
  {% endif %}
{% else %}
  <p>No tienes ningún permiso en esta aplicación.</p>
{% endif %}
```

El resto de detalles: permisos, grupos, mensajes

Hay unas cuantas cosas que pertenecen al entorno de autentificación y que hasta ahora sólo hemos podido ver de pasada. En esta sección las veremos con un poco más de detalle.

Permisos

Los permisos son una forma sencilla de "marcar" que determinados usuarios o grupos pueden realizar una acción. Se usan normalmente para la parte de administración de Django, pero puedes usarlos también en tu código.

El sistema de administración de Django utiliza los siguientes permisos:

- Acceso a visualizar el formulario "Añadir", y Añadir objetos, está limitado a los usuarios que tengan el permiso *add* para ese tipo de objeto.

- El acceso a la lista de cambios, ver el formulario de cambios y cambiar un objeto está limitado a los usuarios que tengan el permisos *change* para ese tipo de objeto.

- Borrar objetos está limitado a los usuarios que tengan el permiso *delete* para ese tipo de objeto.

Los permisos se definen a nivel de las clases o tipos de objetos, no a nivel de instancias. Por ejemplo, se puede decir "María puede modificar los reportajes nuevos", pero no "María solo puede modificar los reportajes nuevos que haya creado ella", ni "María sólo puede cambiar los reportajes que tengan un determinado estado, fecha de publicación o identificador".

Estos tres permisos básicos, añadir, cambiar y borrar, se crean automáticamente para cualquier modelo Django que incluya una clase Admin. Entre bambalinas, los permisos se agregan a la tabla auth_permission cuando ejecutas manage.py migrate. Estos permisos se crean con el siguiente formato: "<app>.<action>_<object_name>". Por ejemplo, si tienes una aplicación llamada encuestas, con un modelo llamado Respuesta, se crearan automáticamente los tres permisos con los nombres "encuestas.add_respuesta", "encuestas.change_respuesta" y "encuestas.delete_respuesta".

Igual que con los usuarios, los permisos se implementa en un modelo Django que reside en el módulo django.contrib.auth.models. Esto significa que puedes usar la API de acceso a la base de datos para interactuar con los permisos de la forma que quieras.

Grupos

Los grupos son una forma genérica de trabajar con varios usuarios a la vez, de forma que se les pueda asignar permisos o etiquetas en bloque. Un usuario puede pertenecer a varios grupos a la vez.

Un usuario que pertenezca a un grupo recibe automáticamente todos los permisos que se la hayan otorgado al grupo. Por ejemplo, si el grupo Editores tiene el permiso can_edit_home_page, cualquier usuario que pertenezca a dicho grupo también tiene ese permiso.

Los grupos también son una forma cómoda de categorizar a los usuarios para asignarles una determinada etiqueta, o para otorgarles una funcionalidad extra. Por ejemplo, se puede crear un grupo Usuarios especiales, y utilizar código para permitir el acceso a determinadas porciones de tu sitio sólo a los miembros de ese grupo, o para enviarles un correo electrónico sólo a ellos.

Al igual que con los usuarios, la manera más sencilla de gestionar los grupos es usando la interfaz de administración de Django. Los grupos, en cualquier caso, son modelos Django que residen en el módulo django.contrib.auth.models así que, al igual que en el caso anterior, puedes usar la API de acceso a la base de datos para trabajar con los grupos a bajo nivel.

Mensajes

El sistema de mensajes es una forma muy ligera y sencilla de enviarle mensajes a un usuario. Cada usuario tiene asociada una cola de mensajes, de forma que los mensajes lleguen en el orden en que fueron enviados. Los mensajes no tienen ni fecha de caducidad ni fecha de envío.

La interfaz de administración de Django usa los mensajes para notificar que determinadas acciones han podido ser llevadas a cabo con éxito. Por ejemplo, al crear un objeto, verás que aparece un mensaje en lo alto de la página de administración, indicando que se ha podido crear el objeto sin problemas.

Puedes usar la misma API para enviar o mostrar mensajes en tu propia aplicación. Las llamadas de la API son bastante simples:

Para crear un nuevo mensaje usa

```
user.message_set.create(message='message_text').
```

Para recuperar/eliminar mensajes usa user.get_and_delete_messages(), la cual retorna una lista de objetos Message en la cola del usuario (si es que existiera alguno) y elimina el mensaje de la misma.

En el siguiente ejemplo, la vista guarda un mensaje para el usuario después de crear una lista de reproducción:

```
def crear_lista(request, songs):
    # Crea un lista de reproducción  con las canciones dadas.
    # ...
    request.user.message_set.create(
        message="Tu lista fue agregada correctamente."
    )
    return render_to_response("playlists/crear.html",
        context_instance=RequestContext(request))
```

Al usar RequestContext, los mensajes del usuario actual, si los tuviera, están accesibles desde la variable de contexto usando el nombre {{ messages }}. El siguiente ejemplo representa un fragmento de código que muestras los mensajes:

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

Hay que hacer notar que RequestContext llama a get_and_delete_messages de forma implícita, por lo que los mensajes serán borrados, aún si no se muestran en pantalla.

Por último, el sistema de mensajería sólo funciona para usuarios de la base de datos. Para enviar mensajes a usuarios anónimos hay que usar en entorno de sesiones directamente.

¿Qué sigue?

La verdad es que el sistema de autorización tiene tela de donde cortar. Sin embargo la mayoría de las veces no tendrás que preocuparte por todos los detalles que se describen en este capítulo, pero si alguna vez tienes que gestionar interacciones complicadas con los usuarios, agradecerás tener a la mano todas estas utilidades disponibles.

En él *próximo capítulo*, echaremos un vistazo a una parte de Django que necesita la infraestructura que proporciona el sistema de cache, el cual es una forma conveniente para mejorar el funcionamiento y rendimiento de tus aplicaciones.

CAPÍTULO 15

El sistema de Cache

Los sitios Web estáticos, en los que las páginas son servidas directamente por el servidor Web, generan un gran escalamiento. Una gran desventaja en los sitios Web dinámicos, es precisamente eso, que son dinámicos. Cada vez que un usuario pide una página, el servidor realiza una serie de cálculos –consultas a una base de datos, renderizado de plantillas, lógica de negocio –para crear la página que el visitante finalmente ve. Esto es costoso desde el punto de vista del sobreprocesamiento.

Para la mayoría de las aplicaciones Web, esta sobrecarga no es gran cosa. La mayoría de las aplicaciones Web no son el Washingtonpost o Pinterest; son de un tamaño pequeño a uno mediano, y con poco tráfico. Pero para los sitios con tráfico de medio a alto es esencial bajar lo más que se pueda el costo de procesamiento. He aquí cuando realizar un cache es de mucha ayuda.

Colocar en cache algo significa guardar el resultado de un cálculo costoso para que no se tenga que realizar el mismo la próxima vez. Aquí mostramos un pseudocódigo explicando cómo podría funcionar esto para una página Web dinámica:

```
dada una URL, buscar esa página en la cache

si la página está en la cache:
    devolver la página en cache

si no:
    generar la página
    guardar la página generada en la cache (para la próxima vez)
    devolver la página generada
```

Django incluye un sistema de cache robusto que permite guardar páginas dinámicas para que no tengan que ser recalculadas cada vez que se piden. Por conveniencia, Django ofrece diferentes niveles de granularidad de cache. Puedes dejar en cache el resultado de diferentes vistas, sólo las piezas que son difíciles de producir, o se puede dejar en cache el sitio entero.

Django también trabaja muy bien con caches de "downstream", tales como Squid y las caches de los navegadores. Estos son los tipos de cache que no controlas directamente pero a las cuales puedes proveerles algunas pistas (vía cabeceras HTTP) acerca de qué partes de tu sitio deben ser colocadas en cache y cómo.

Sigue leyendo para descubrir cómo usar el sistema de cache de Django. Cuando tu sitio se parezca cada vez más a Pinterest, estarás contento de entender este material.

Activar la Cache

El sistema de cache requiere sólo una pequeña configuración. A saber, tendrás que decirle donde vivirán los datos de tu cache, si es en una base de datos, en el sistema de archivos, o directamente en memoria. Esta es una decisión importante que afecta el rendimiento de la cache (si, algunos tipos de cache son más rápidos que otros). La cache en memoria generalmente será mucho más rápida que la cache en el sistema de archivos o la cache en una base de datos, porque carece del trabajo de tocar los mismos.

Tus preferencias acerca de la cache van en la variable CACHE en el archivo de configuración. A continuación daremos un recorrido por todos los valores y configuraciones disponibles que puedes usar para configurar la CACHE.

Memcached

Por mucho, el más rápido y eficiente soporte nativo de cache para Django es memcached, el cual es un framework de cache basado enteramente en memoria, originalmente desarrollado para manejar grandes cargas en LiveJournal (http://www.livejournal.com/)y subsecuentemente por Danga Interactive (http://danga.com/). Es usado por sitios como Slashdot y Wikipedia para reducir el acceso a bases de datos e incrementar el rendimiento dramáticamente.

Memcached está disponible libremente para descargar. Corre como un demonio y se le asigna una cantidad específica de memoria RAM. Su característica principal es proveer una interfaz – *super-liviana-y-rápida* para añadir, obtener y eliminar arbitrariamente datos en la cache. Todos los datos son guardados directamente en memoria, por lo tanto no existe sobrecarga de uso en una base de datos o en el sistema de archivos.

Después de haber instalado Memcached, es necesario que instales alguno de los adaptadores disponibles para usar Python con Memcached, los cuales no vienen incluidas con Django.

Dichos *adaptadores* pueden ser python-memcached y/o pylibmc. Los cuales están disponibles como módulos de Python.

Para usar Memcached con Django, usa como BACKEND a: django.core.cache.backends.memcached.MemcachedCache o django.core.cache.backends.memcached.PyLibMCCache (Dependiendo de el adaptador que hayas elegido usar). Fija el valor de LOCATION como ip:puerto, donde ip es la dirección IP del demonio de Memcached y puerto es el puerto donde Memcached está corriendo o usa los valores unix:path, donde path es la ruta al archivo usado como socket en unix por Memcached.

En el siguiente ejemplo, Memcached está corriendo en localhost (127.0.0.1) en el puerto 11211, usando como dependencia python-memcached:

```
CACHES = {
   'default': {
      'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
      'LOCATION': '127.0.0.1:11211',
   }
}
```

En el siguiente ejemplo, Memcache está disponible a través del socket local unix, que usa el archivo /tmp/memcached.sock como socket, usando los enlaces proporcionados por python-memcached:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}
```

Una muy buena característica de Memcached es su habilidad de compartir la cache en varios servidores. Esto significa que puedes correr demonios de Memcached en diferentes máquinas, y el programa seguirá tratando el grupo de diferentes máquinas como una *sola* cache, sin la necesidad de duplicar los valores de la cache en cada máquina. Para sacar provecho de esta característica con Django, incluye todas las direcciones de los servidores en LOCATION, separados por punto y coma.

En el siguiente ejemplo, la cache es compartida en varias instancias de Memcached en las direcciones IP 172.19.26.240 y 172.19.26.242, ambas en el puerto 11211:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
            ]
    }
}
```

En el siguiente ejemplo, la cache es compartida en diferentes instancias de Memcached corriendo en las direcciones IP 172.19.26.240 (puerto 11211), 172.19.126.242 (puerto 11212) y 172.19.26.244 (puerto 11213):

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11212',
            '172.19.26.244:11213',
          ]
    }
}
```

Una última observación acerca de Memcached es que la cache basada en memoria tiene una importante desventaja. Como los datos de la cache son guardados en memoria, serán perdidos si los servidores se caen. Más claramente, la memoria no es para almacenamiento permanente, por lo tanto no te quedes solamente con una cache basada en memoria. Sin duda, *ninguno* de los sistemas de cache de Django debe ser utilizado para almacenamiento permanente –son todos una solución para la cache, no para almacenamiento pero hacemos hincapié aquí porque la cache basada en memoria es únicamente para uso temporal.

Cache en Base de datos

Para usar una tabla de una base de datos como cache, tienes que crear una tabla en tu base de datos y apuntar el sistema de cache de Django a ella.

Primero, crea la tabla de cache corriendo el siguiente comando:

```
python manage.py createcachetable [nombre_tabla_cache]
```

Donde [nombre_tabla_cache] es el nombre de la tabla a crear. Este nombre puede ser cualquiera que desees, siempre y cuando sea un nombre válido para una tabla y que no esté ya en uso en tu base de datos. Este comando crea una única tabla en tu base de datos con un formato apropiado para el sistema de cache de Django.

Una vez que se hayas creado la tabla, usa la propiedad LOCATION como LOCATION:nombre_tabla, donde nombre_tabla es el nombre de la tabla en la base de datos y usa como BACKEND django.core.cache.backends.db.DatabaseCache
En el siguiente ejemplo, el nombre de la tabla para el cache es mi_tabla_cache:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'mi_tabla_cache',
    }
}
```

El sistema de cache usará la misma base de datos especificada en el archivo de configuración. Por lo que no podrás usar una base de datos diferente, a menos que la registres primero.

Cache en Sistema de Archivos

Para almacenar la cache en el sistema de archivos y almacenar cada valor de la cache como un archivo separado, configura la propiedad BACKEND usando django.core.cache.backends.filebased.FileBasedCache y especificando en LOCATION el directorio en tu sistema de archivos que debería almacenar los datos de la cache.

Por ejemplo, para almacenar los datos de la cache en /var/tmp/django_cache, coloca lo siguiente:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

Si usas Windows, especifica la letra de la unidad al comienzo de la ruta de directorios de esta forma:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': 'c:/usuarios/temp',
    }
}
```

La ruta de directorios, debe ser *absoluta* –debe comenzar con la raíz de tu sistema de archivos. No importa si colocas una barra al final de la misma.

Asegúrate que el directorio apuntado por esta propiedad exista y que pueda ser leído y escrito por el usuario del sistema usado por tu servidor Web, para ejecutarse.

Continuando con el ejemplo anterior, si tu servidor corre como usuario apache, asegúrate que el directorio /var/tmp/django_cache exista y pueda ser leído y escrito por el usuario apache.

Cada valor de la cache será almacenado como un archivo separado conteniendo los datos de la cache serializados ("pickled"), usando el módulo Python pickle. Cada nombre de archivo es una clave de la cache, modificado convenientemente para que pueda ser usado por el sistema de archivos.

Cache en Memoria local

Si quieres usar la ventaja que otorga la velocidad de la cache en memoria, pero no tienes la capacidad de correr Memcached, puedes optar por el cache de memoria-local. Esta cache es por proceso y usa hilos-seguros, pero no es tan eficiente como Memcache dada su estrategia de bloqueo simple y reserva de memoria.

Para usarla, usa como BACKEND a django.core.cache.backends.locmem.LocMemCache. Por ejemplo:

```
CACHES = {
   'default': {
      'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
      'LOCATION': 'unico-proceso'
   }
}
```

El atributo LOCATION de la cache es usada para identificar de forma individual el almacenamiento de la memoria. Si utilizas únicamente un proceso puedes omitir LOCATION; sin embargo si utilizas más de uno, necesitas asignar un nombre a al menos uno de los procesos para mantenerlos separados.

Observa que cada proceso tendrá su propia instancia de cache privada, lo cual significa que no es posible el proceso cruzado de cache. Esto obviamente también significa que la memoria local de cache no es particularmente muy eficiente, así que no es una buena opción para usar en ambientes de producción. Es recomendable solo para desarrollo.

Cache personalizada

A pesar de que Django incluye soporte para el uso de un buen número de sistemas de cache fuera de la caja, algunas veces puede que quieras usar un almacenamiento de cache personalizado, para fines específicos.

Para usar almacenamiento externo de cache con Django, usa la ruta de importaciones de Python como BACKEND y carga la configuración de la cache así:

```
CACHES = {
   'default': {
      'BACKEND': 'ruta.a.backend',
   }
}
```

Si estas construyendo tu propio sistema de cache, puedes usar el sistema de almacenamiento de caches de Django como referencia para implementar el tuyo.

Puedes encontrar el código fuente en el directorio ubicado en: django/core/cache/backends/

Cache tonta (para desarrollo)

Finalmente, Django incluye una cache tonta formalmente llamada: "dummy" que no realiza cache; sólo implementa la interfaz de cache sin realizar ninguna acción.

Esta es útil cuando tienes un sitio en producción que usa mucho cache en varias partes y en un entorno de desarrollo/prueba en el cual no quieres hacer cache. En ese caso, usa BACKEND como django.core.cache.backends.dummy.DummyCache en el archivo de configuración para tu entorno de desarrollo, por ejemplo:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

Como resultado de esto, tu entorno de desarrollo no usará cache, pero tu entorno de producción si lo hará.

Argumentos de cache

Cada tipo de cache puede recibir argumentos adicionales para controlar el comportamiento de la cache. Estos son dados como una clave adicional a la configuración de CACHES. Los argumentos válidos son los siguientes:

- **TIMEOUT**: El tiempo de vida por omisión, en segundos, que usará la cache. Este argumento tomará el valor de 300 segundos (5 minutos) si no se lo especifica.

  También puedes especificar TIMEOUT como None, por defecto la clave de la cache nunca expira.

- **OPTIONS**: Cualquier opción que se necesite pasar a la cache. La lista de opciones validas dependerá de cada backend, por lo que el almacenamiento de cache proporcionado por librerías de terceros, será pasado con sus opciones directamente bajo la cache de la librería.

  Los almacenamientos de cache que implementan sus propias estrategias de selección (por ejemplo: en memoria, archivos y en base de datos) respetan las siguientes opciones:

- **MAX_ENTRIES**: Para la cache de memoria local, y la cache de base de datos, es el número máximo de entradas permitidas en la cache a partir del cual los valores más viejos serán eliminados. Tomará un valor de 300 si no se lo especifica.

- **CULL_FREQUENCY:** La proporción de entradas que serán sacrificadas cuando la cantidad de MAX_ENTRIES es alcanzada. La proporción real es 1/CULL_FREQUENCY, si quieres sacrificar la mitad de las entradas cuando se llegue a una cantidad de MAX_ENTRIES coloca CULL_FREQUENCY=2. Este argumento tomará un valor de 3 si no se especifica.

Un valor de 0 para CULL_FREQUENCY significa que toda la cache será limpiada cuando se llegue a una cantidad de entradas igual a MAX_ENTRIES. Esto hace que el proceso de limpieza de la cache sea *mucho* más rápido, con el costo de perder más datos de la cache. Este argumento tomará un valor de 3 si no se especifica.

- **KEY_PREFIX:** Una cadena que automáticamente incluye (agrega por default) todas las claves de caches usadas por el servidor Django.

- **VERSION** El número de versión de las claves de cache generadas por el servidor Django.

- **KEY_FUNCTION**: Una cadena que contiene la ruta (usando el punto) a la función que define la forma en que está compuesta el prefijo, la versión y la clave, en la clave de la cache final.

En este ejemplo, usamos un "archivo" como almacenamiento de cache (BACKEND), configurado con un valor de tiempo de 60 segundos (TIMEOUT) y con una capacidad máxima (MAX_ENTRIES) de 1000 ítems:

```
CACHES = {
   'default': {
   'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
      'LOCATION': '/var/tmp/django_cache',
      'TIMEOUT': 60,
      'OPTIONS': {
         'MAX_ENTRIES': 1000
      }
   }
}
```

Tanto los argumentos desconocidos, así como los valores inválidos de argumentos conocidos son ignorados silenciosamente.

La cache por sitio

Una vez que hayas especificado CACHE, la manera más simple de usar la cache es colocar en cache el sitio entero. Esto significa que cada página que no tenga parámetros GET o POST será puesta en cache por un cierto período de tiempo la primera vez que sean pedidas.

Para activar la cache por sitio agrega:
'django.middleware.cache.CacheMiddleware' y
django.middleware.cache.FetchFromCacheMiddleware a la propiedad
MIDDLEWARE_CLASSES, como en el siguiente ejemplo:

```
MIDDLEWARE_CLASSES = (
   'django.middleware.cache.UpdateCacheMiddleware',
   'django.middleware.common.CommonMiddleware',
   'django.middleware.cache.FetchFromCacheMiddleware',
)
```

Nota: El orden de MIDDLEWARE_CLASSES importa. Mira la sección "Orden de MIDDLEWARE_CLASSES" más adelante en este capítulo.

Luego, agrega las siguientes propiedades en el archivo de configuración de Django:

- **CACHE_MIDDLEWARE_ALIAS**: El nombre del alias para usar como almacenaje.
- **CACHE_MIDDLEWARE_SECONDS**: El tiempo en segundos que cada página será mantenida en la cache.
- **CACHE_MIDDLEWARE_KEY_PREFIX:** Si la cache es compartida a través de múltiples sitios usando la misma instalación Django, coloca esta propiedad como el nombre del sitio, u otra cadena que sea única para la instancia de Django, para prevenir colisiones. Usa una cadena vacía si no te interesa.

La cache middleware coloca en cache cada página que no tenga parámetros GET o POST. Esto significa que si un usuario pide una página y pasa parámetros GET en la cadena de consulta, o pasa parámetros POST, la cache middleware *no* intentará obtener la versión en cache de la página. Si intentas usar la cache por sitio ten esto en mente cuando diseñes tu aplicación; no uses URLs con cadena de consulta, por ejemplo, a menos que sea aceptable que tu aplicación no coloque en cache esas páginas.

Finalmente, nota que CacheMiddleware automáticamente coloca unos pocos encabezados en cada HttpResponse:

- Coloca el encabezado *Last-Modified* con el valor actual de la fecha y hora cuando una página (aún no en cache) es requerida.
- Coloca el encabezado *Expires* con el valor de la fecha y hora más el tiempo definido en CACHE_MIDDLEWARE_SECONDS.
- Coloca el encabezado *Cache-Control* para otorgarle una vida máxima a la página, como se especifica en CACHE_MIDDLEWARE_SECONDS.

Cache para vistas

Una forma más granular de usar el framework de cache es colocar en cache la salida de las diferentes vistas. Esto tiene el mismo efecto que la cache por sitio (incluyendo la omisión de colocar en cache los pedidos con parámetros GET y POST). Se aplica a cualquier vista que tú especifiques, en vez de aplicarse al sitio entero.

Haz esto usando un *decorador*, que es un wrapper de la función de la vista que altera su comportamiento para usar la cache. El decorador de cache por vista es llamado cache_page y se encuentra en el módulo django.views.decorators.cache, por ejemplo:

```
from django.views.decorators.cache import cache_page

def mi_vista(request, param):
    # ...
    mi_vista = cache_page(mi_vista, 60 * 15)
```

De otra manera, si estás usando alguna versión de Python, superior a la 2.7, puedes usar un decorador. El siguiente ejemplo es equivalente al anterior:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def mi_vista(request, param):

# ...
```

cache_page recibe un único argumento: el tiempo de vida en segundos de la cache. En el ejemplo anterior, el resultado de mi_vista() estará en cache unos 15 minutos. (Toma nota de que lo hemos escrito como 60 * 15 para que sea entendible. 60 * 15 será evaluado como 900 –que es igual a 15 minutos multiplicados por 60 segundos cada minuto.)

La cache por vista, como la cache por sitio, es indexada independientemente de la URL. Si múltiples URLs apuntan a la misma vista, cada URL será puesta en cache separadamente.

Continuando con el ejemplo de mi_vista, si tu URLconf se ve como:

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', mi_vista),
]
```

Los pedidos a /foo/1/ y a /foo/23/ serán puestos en cache separadamente, como es de esperar. Pero una vez que una misma URL es pedida (p.e. /foo/23/), los siguientes pedidos a esa URL utilizarán la cache.

cache_page toma un argumento de clave opcional: llamado cache, el cual puede usarse directamente en el decorador especificando la cache (tomada de el archivo de configuración de la variable CACHE) para cachear la vista. Por defecto, el cache para usar será especificado con cualquier cache que queramos, por ejemplo:

```
@cache_page(60 * 15, cache="cache_especial")

def mi_vista(request):
    ...
    algun_metodo()
```

También es posible sobrescribir el prefijo de la cache en la vista. El decorador cache_page toma un argumento de clave ***key_prefix,*** el cual trabaja de la misma forma que la configuración CACHE_MIDDLEWARE_KEY_PREFIX en el middleware. Puede usarse de la siguiente forma:

```
@cache_page(60 * 15, key_prefix="sitio1")

def mi_vista(request):
    ...
    algun_metodo()
```

Las dos configuraciones pueden ser combinadas. Si especificas cache y key_prefix puedes traer todas las configuraciones en la petición usando alias en la cache, solo que esto sobrescribirá el argumento key_prefix.

Cache por vista en la URLconf

Los ejemplos en la sección anterior incrustan la cache en las vistas, porque el decorador cache_page modifica la función mi_vista en la misma vista. Este enfoque acopla tu vista con el sistema de cache, lo cual no es lo ideal por varias razones. Por ejemplo, puede que quieras rehusar las funciones de la vista en otro sitio sin cache, o puede que quieras distribuir las vistas a gente que quiera usarlas sin que sean colocadas en la cache. La solución para estos problemas es especificar la cache por vista en URLconf en vez de especificarla junto a las vistas mismas.

Hacer eso es muy fácil: simplemente envuelve la función de la vista con cache_page cuando hagas referencia a ella en la URLconf. Aquí el URLconf como estaba antes:

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', mi_vista),
]
```

Ahora la misma cosa con la función mi_vista usando el decorador cache_page:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(mi_vista)),
]
```

Si tomas este enfoque no olvides de importar cache_page dentro de tu URLconf.

La API de cache de bajo nivel

Algunas veces, colocar en cache una página entera no te hace ganar mucho y es, de hecho, un inconveniente excesivo.

Quizás, por ejemplo, tu sitio incluye una vista cuyos resultados dependen de diversas consultas costosas, lo resultados de las cuales cambian en intervalos diferentes. En este caso, no sería ideal usar la página entera en cache que la cache por sitio o por vista ofrecen, porque no querrás guardar en cache todo el resultado (ya que los resultados cambian frecuentemente), pero querrás guardar en cache los resultados que rara vez cambian.

Para casos como este, Django expone una simple API de cache de bajo nivel, la cual vive en el módulo django.core.cache. Puedes usar la API de cache de bajo nivel para almacenar los objetos en la cache con cualquier nivel de granularidad que te guste. Puedes colocar en la cache cualquier objeto Python que pueda ser serializado de forma segura: strings, diccionarios, listas de objetos del modelo, y demás. (La mayoría de los objetos comunes de Python pueden ser serializados; revisa la documentación de Python para más información acerca de serialización).

Aquí vemos como importar la API:

```
>>> from django.core.cache import cache
```

La interfaz básica es set(key, value, timeout) y get(key):

```
>>> cache.set('mi_clave', '¡Hola Mundo!', 30)
>>> cache.get('mi_clave')
' ¡Hola Mundo!'
```

El argumento timeout es opcional y obtiene el valor del argumento timeout de la variable CACHE, explicado anteriormente, si no se lo especifica.

Si el objeto no existe en la cache, o el sistema de cache no se puede alcanzar, cache.get() devuelve None:

```
# Espera  30 segundos por 'mi_clave'  para que expire...
>>> cache.get('mi_clave')
None
>>> cache.get('otra_clave')
None
```

Te recomendamos que no almacenes el valor literal None en la cache, porque no podrás distinguir entre tu valor None almacenado y el valor que devuelve la cache cuando no encuentra un objeto.

cache.get() puede recibir un argumento por omisión. Esto especifica qué valor debe devolver si el objeto no existe en la cache:

```
>>> cache.get('mi_clave', 'ha expirado')
'ha expirado'
```

Para obtener múltiples valores de la cache de una sola vez, usa cache.get_many(). Si al sistema de cache le es posible, get_many() tocará la cache sólo una vez, al contrario de tocar la cache por cada valor. get_many() devuelve un diccionario con todas las claves que has pedido que existen en la cache y todavía no han expirado:

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
```

Si una clave no existe o ha expirado, no será incluida en el diccionario. Lo siguiente es una continuación del ejemplo anterior:

```
>>> cache.get_many(['a', 'b', 'c', 'd'])
{'a': 1, 'b': 2, 'c': 3}
```

Finalmente, puedes eliminar claves explícitamente con cache.delete(). Esta es una manera fácil de limpiar la cache para un objeto en particular:

```
>>> cache.delete('a')
```

cache.delete() no tiene un valor de retorno, y funciona de la misma manera si existe o no un valor en la cache.

Caches downstream

Este capítulo se ha enfocado en la cache de tus *propios* datos. Pero existe otro tipo de cache que es muy importante para los desarrolladores web: la cache realizada por los *downstream*. Estos son sistemas que colocan en cache páginas aún antes de que estas sean pedidas a tu sitio Web.

Aquí hay algunos ejemplos de caches para downstream:

- Tu ISP puede tener en cache algunas páginas, si tu pides una página de http://example.com/, tu ISP te enviará la página sin tener que acceder a example.com directamente. Los responsables de example.com no tienen idea que esto pasa; el ISP se coloca entre example.com y tu navegador, manejando todo lo que se refiera a cache transparentemente.

- Tu sitio en Django puede colocarse detrás de un *cache proxy*, como Squid Web Proxy Cache (🌎 http::://www.squid-cache.org/), que coloca en cache páginas para un mejor rendimiento. En este caso, cada pedido será controlado por el proxy antes que nada, y será pasado a tu aplicación sólo si es necesario.

- Tu navegador también pone páginas en un cache. Si una página Web envía unos encabezados apropiados, tu navegador usará su copia de la cache local para los siguientes pedidos a esa página, sin siquiera hacer nuevamente contacto con la página web para ver si esta ha cambiado.

La cache de downstream es un gran beneficio, pero puede ser peligroso. El contenido de muchas páginas Web pueden cambiar según la autenticación que se haya realizado u otras variables, y los sistemas basados en almacenar en cache según la URL pueden exponer datos incorrectos o delicados a diferentes visitantes de esas páginas.

Por ejemplo, digamos que manejas un sistema de e-mail basado en Web, el contenido de la "bandeja de entrada" obviamente depende de que usuario esté logueado. Si el ISP hace caching de tu sitio ciegamente, el primer usuario que ingrese al sistema compartirá su bandeja de entrada, que está en cache, con los demás usuarios del sistema. Eso, definitivamente no es bueno.

Afortunadamente, el protocolo HTTP provee una solución a este problema. Existen un número de encabezados HTTP que indican a las cache de downstream que diferencien sus contenidos de la cache dependiendo de algunas variables, y para que algunas páginas particulares no se coloquen en cache. Veremos algunos de estos encabezados en las secciones que siguen.

Usar el encabezado Vary

El encabezado Vary define cuales encabezados debería tener en cuenta un sistema de cache cuando construye claves de su cache. Por ejemplo, si el contenido de una página Web depende de las preferencias de lenguaje del usuario, se dice que la página "varía según el lenguaje".

Por omisión, el sistema de cache de Django crea sus claves de cache usando la ruta que se ha requerido (p.e.: "/stories/2005/jun/23/bank_robbed/"). Esto significa que cada pedido a esa URL usará la misma versión de cache, independientemente de las características del navegador del cliente, como las cookies o las preferencias del lenguaje. Sin embargo, si esta página produce contenidos diferentes basándose en algunas cabeceras del request–como las cookies, el lenguaje, o el navegador–

necesitarás usar el encabezado Vary para indicarle a la cache que esa página depende de esas cosas.

Para hacer esto en Django, usa el decorador vary_on_headers como sigue:

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def mi_vista(request):
    # ...
```

En este caso, el mecanismo de cache (como middleware) colocará en cache una versión distinta de la página para cada tipo de user-agent.

La ventaja de usar el decorador vary_on_headers en vez de fijar manualmente el encabezado Vary (usando algo como response['Vary'] = 'user-agent') es que el decorador *agrega* al encabezado Vary (el cual podría ya existir), en vez de fijarlo desde cero y potencialmente sobrescribir lo que ya había ahí.

Puedes pasar múltiples encabezados a vary_on_headers():

```
@vary_on_headers('User-Agent', 'Cookie')
def mi_vista(request):
    # ...
```

Esto le dice a la cache de downstream que diferencie *ambos*, lo que significa que cada combinación de una cookie y un navegador obtendrán su propio valor en cache. Por ejemplo, un pedido con navegador Mozilla y una cookie con el valor foo=bar será considerada diferente a un pedido con el navegador Mozilla y una cookie con el valor foo=ham.

Como las variaciones con las cookies son tan comunes existe un decorador vary_on_cookie. Las siguientes dos vistas son equivalentes:

```
@vary_on_cookie
def mi_vista(request):
    # ...

@vary_on_headers('Cookie')
def mi_vista(request):
    # ...
```

El encabezado que le pasas a vary_on_headers no diferencia mayúsculas de minúsculas; "User-Agent" es lo mismo que "user-agent".

También puedes usar django.utils.cache.patch_vary_headers como función de ayuda. Esta función fija o añade al Vary header, por ejemplo:

```
from django.utils.cache import patch_vary_headers

def mi_vista(request):
    # ...
    response = render_to_response('template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

patch_vary_headers obtiene una instancia de HttpResponse como su primer argumento y una lista/tupla de nombres de encabezados, sin diferenciar mayúsculas de minúsculas, como su segundo argumento.

Controlando el cache: usando otros Encabezados

Otro problema con la cache es la privacidad de los datos y donde deberían almacenarse los datos cuando se hace un vuelco de la cache.

El usuario generalmente se enfrenta con dos tipos de cache: su propia cache de su navegador (una cache privada) y la cache de su proveedor (una cache pública). Una cache pública es usada por múltiples usuarios y controlada por algunos otros. Esto genera un problema con datos sensibles–no quieres que, por ejemplo, el número de tu cuenta bancaria sea almacenado en una cache pública. Por lo que las aplicaciones Web necesitan una manera de indicarle a la cache cuales datos son privados y cuales son públicos.

La solución es indicar que la copia en cache de una página es "privada". Para hacer esto en Django usa el decorador de vista cache_control:

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def mi_vista(request):
    # ...
```

Este decorador se encarga de enviar los encabezados HTTP apropiados detrás de escena.

Nota que el control de configuraciones de cache "privado" y "publico" es mutuamente excluyente. El decorador se asegura que la directiva "publico" sea removida si se encuentra configurado como "privado" (y viceversa). Un ejemplo del uso de estas dos directivas, puede ser un sitio de un blog que ofrece entradas públicas y privadas. Las entradas públicas pueden ser cacheadas en la cache compartida. El siguiente código usa django.utils.cache.patch_cache_control() para manualmente modificar el control de las cabeceras de la cache (Es internamente llamado por el decorador cache_control).

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def lista_de entradas_blog(request):
   if request.user.is_anonymous():
      response = render_only_public_entries()
      patch_cache_control(response, public=True)
   else:
      response = render_private_and_public_entries(request.user)
      patch_cache_control(response, private=True)

      return response
```

Existen otras pocas maneras de controlar los parámetros de cache. Por ejemplo, HTTP permite a las aplicaciones hacer lo siguiente:

- Definir el tiempo máximo que una página debe estar en cache.
- Especificar si una cache debería comprobar siempre la existencia de nuevas versiones, entregando únicamente el contenido de la cache cuando no hubiesen cambios. (Algunas caches pueden entregar contenido aun si la

página en el servidor ha cambiado, simplemente porque la copia en cache todavía no ha expirado.)

Django, utiliza el decorador cache_control para especificar estos parámetros de la cache. En el siguiente ejemplo, cache_control le indica a la cache revalidarse en cada acceso y almacenar versiones en cache hasta por 3.600 segundos:

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True, max_age=3600)
def mi_vista(request):
    ...
```

Cualquier directiva Cache-Control de HTTP válida es válida en cache_control(). Aquí hay una lista completa:

- public=True
- private=True
- no_cache=True
- no_transform=True
- must_revalidate=True
- proxy_revalidate=True
- max_age=num_seconds
- s_maxage=num_seconds

■**Tip:** Para una explicación de las directivas Cache-Control de HTTP, consulta las especificaciones en http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9.

■**Nota:** El middleware de caching ya fija el encabezado max-age con el valor de CACHE_MIDDLEWARE_SETTINGS. Si utilizas un valor propio de max_age en un decorador cache_control, el decorador tendrá precedencia, y los valores del encabezado serán fusionados correctamente.

Si quieres usar cabeceras para desactivar el cache por completo, usa el decorador never_cache del paquete django.views.decorators.cache.never_cache en una vista, a la que le quieras agregar la cabecera, para asegurarte que la respuesta no sea cacheada por el navegador u otros caches. Por ejemplo:

```
from django.views.decorators.cache import never_cache

@never_cache
def mi_vista(request):
    # ...
```

Otras optimizaciones

Django incluye otras piezas de middleware que pueden ser de ayuda para optimizar el rendimiento de tus aplicaciones:

- django.middleware.http.ConditionalGetMiddleware agrega soporte para navegadores modernos para condicionar respuestas GET basadas en los encabezados ETag y Las-Modified.

- django.middleware.gzip.GZipMiddleware comprime las respuestas para todos los navegadores modernos, ahorrando ancho de banda y tiempo de transferencia.

Orden de MIDDLEWARE_CLASSES

Si utilizas CacheMiddleware, es importante colocarlas en el lugar correcto dentro de la propiedad MIDDLEWARE_CLASSES, porque el middleware de cache necesita conocer los encabezados por los cuales cambiar el almacenamiento en la cache.

Coloca el CacheMiddleware después de cualquier middleware que pueda agregar algo al encabezado Vary, incluyendo los siguientes:

- *SessionMiddleware,* que agrega Cookie

- *GZipMiddleware*, que agrega Accept-Encoding

- *LocaleMiddleware* que agrega Accept-Language

¿Qué sigue?

Django incluye un número de paquetes opcionales. Hemos cubierto algunos de los mismos: como el sistema de administración en el *capítulo 6*, el marco de sesiones/usuarios del *capítulo 14*.

Él *próximo capítulo* cubre el resto de paquetes incluidos en el modulo “contrib”, que provee una cantidad interesante de herramientas disponibles; que pueden hacer más fácil tu vida, no querrás perderte ninguno de ellos.

CAPÍTULO 16

El paquete django.contrib

Una de las muchas fortalezas de Python, es su filosofía de **baterías incluidas**. Cuando instalas Python, este viene con una amplia biblioteca de paquetes que puedes comenzar a usar inmediatamente, sin necesidad de descargar nada más.

Django trata de seguir esa misma filosofía, e incluye su propia biblioteca estándar de paquetes útiles para realizar las tareas más comunes del desarrollo web. Este capítulo cubre la colección de agregados, agrupados en el paquete llamado django.contrib.

La biblioteca estándar de Django

La biblioteca estándar de Django se localiza en el paquete django.contrib. Dentro de cada sub-paquete hay una pieza aislada de funcionalidad lista para agregar. Estas piezas no están necesariamente relacionadas, pero algunos sub-paquetes de django.contrib requieren de otros paquetes.

No hay grandes requerimientos para los tipos de funcionalidad que hay en django.contrib. Algunos de los paquetes incluyen modelos (y por lo tanto requieren que instales sus tablas en tu base de datos), otros consisten solamente de *middleware* o de etiquetas de plantillas (*template tags*).

La única característica en común de todos los paquetes de django.contrib es la siguiente: si borraras dicho paquete por completo, seguirías pudiendo usar las capacidades fundamentales de Django sin problemas. Cuando los desarrolladores de Django agregan una nueva funcionalidad al *framework*, emplean esa regla de oro al decidir en dónde va a residir la nueva funcionalidad, si en django.contrib, o en algún otro lugar.

django.contrib consiste de los siguientes paquetes:

- **admin:** el sitio automático de administración. Consulta el capítulo 6.
- **admindocs:** La auto documentación para el sitio administrativo.
- **auth**: el framework de autenticación de Django. Consulta el capítulo 14.
- **contenttypes**: un framework para conectar "tipos" de contenido, en que cada modelo de Django instalado es un tipo de contenido aislado. Este framework es usado internamente por otras aplicaciones "contrib", y está especialmente enfocada a los desarrolladores de Django muy avanzados. Dichos desarrolladores pueden hallar más información sobre esta aplicación, leyendo el código fuente que está en django/contrib/contenttypes/.

- **csrf:** protección ante un ataque de falsificación de petición en sitios cruzados, en inglés Cross-Site Request Forgery (CSRF). Consulta la sección titulada "Protección contra CSRF" más adelante.

- **flatpages**: un framework para administrar contenido HTML simple, "plano", dentro de la base de datos. Consulta la sección titulada "Flatpages" más adelante.

- **humanize:** un conjunto de filtros de plantillas Django, útiles para darle un "toque de humanidad" a los datos. Consulta la sección titulada "Humanizando datos" más adelante.

- **gis**: Extencion para Django que provee soporte para GIS (Sistema de Información Geográfica). Permite a los modelos de Django, por ejemplo almacenar datos geográficos y optimizar consultas geográficas. Esta es una larga y compleja librería, consulta mas detalles en: http://geodjango.org/

- **redirects:** un framework para administrar redirecciones. Consulta la sección titulada "Redirects" más adelante.

- **sessions**: el framework de sesiones de Django. Consulta el capítulo 12.
- sitemaps: un framework para generara archivos de mapas de sitio XML. Consulta él capítulo 13.

- **sites:** un framework que te permite operar múltiples sitios web desde la misma base de datos, y con una única instalación de Django.

- **syndication**: un framework para generar documentos de sindicación (feeds), en RSS y en Atom. Consulta él capítulo 13.

- **webdesign**: Agregados en Django, particularmente útiles en el diseño Web más que en el desarrollo. Este incluye únicamente una etiqueta {% lorem %}. que permite escribir texto en las plantillas.

El resto de este capítulo entra en los detalles de cada paquete django.contrib que no ha sido cubierto aún en este libro.

Sites

El sistema *sites* (sitios) de Django es un *framework* genérico que te permite operar múltiples sitios Web desde la misma base de datos, y desde el mismo proyecto de Django. Éste es un concepto abstracto, y puede ser difícil de entender, así que comenzaremos mostrando algunos escenarios en donde sería útil usarlo.

Escenario 1: rehusó de datos en múltiples sitios

Como explicamos en capítulos anteriores, los sitios LJWorld.com y Lawrence.com, que funcionan gracias a Django, son operados por la misma organización de prensa, el diario *Lawrence Journal-World* de Lawrence, Kansas. LJWorld.com se enfoca en noticias, mientras que Lawrence.com se enfoca en el entretenimiento local. Pero a veces los editores quieren publicar un artículo en *ambos* sitios.

La forma tonta de resolver el problema sería usar una base de datos para cada sitio, y pedirle a los productores que publiquen la misma nota dos veces: una para LJWorld.com y nuevamente para Lawrence.com. Pero esto es ineficiente para los productores del sitio, y es redundante conservar múltiples copias de la misma nota en las bases de datos.

¿Una solución mejor? Que ambos sitios usen la misma base de datos de artículos, y que un artículo esté asociado con uno o más sitios por una relación de muchos-a-muchos. El *framework sites* de Django, proporciona la tabla de base de datos que hace que los artículos se puedan relacionar de esta forma. Sirve para asociar datos con uno o más "sitios".

Escenario 2: alojamiento del nombre/dominio de tu sitio en un solo lugar

Los dos sitios LJWorld.com y Lawrence.com, tienen la funcionalidad de alertas por correo electrónico, que les permite a los lectores registrarse para obtener notificaciones. Es bastante básico: un lector se registra en un formulario web, e inmediatamente obtiene un correo electrónico que dice "Gracias por su suscripción".

Sería ineficiente y redundante implementar el código del procesamiento de registros dos veces, así que los sitios usan el mismo código detrás de escena. Pero la noticia "Gracias por su suscripción" debe ser distinta para cada sitio, empleando objetos Site, podemos abstraer el agradecimiento para usar los valores del nombre y dominio del sitio, variables name (ejemplo: 'LJWorld.com') y domain (ejemplo: 'www.ljworld.com').

El *framework sites* te proporciona un lugar para que puedas almacenar el nombre (name) y el dominio (domain) de cada sitio de tu proyecto, lo que significa que puedes reutilizar estos valores de manera genérica.

Como usar el framework sites

Sites más que un *framework*, es una serie de convenciones. Toda la cosa se basa en dos conceptos simples:

- El modelo Site, que se halla en django.contrib.sites, tiene los campos domain y name.
- La opción de configuración SITE_ID especifica el ID de la base de datos del objeto Site asociado con este archivo de configuración en particular.

La manera en que uses estos dos conceptos queda a tu criterio, pero Django los usa de varios modos de manera automática, siguiendo convenciones simples.

Para instalar la aplicación *sites*, sigue estos pasos:

1. Agrega 'django.contrib.sites' a tu INSTALLED_APPS.
2. Ejecuta el comando `python manage.py migrate` para instalar la tabla django_site en tu base de datos.
3. Agrega uno o más objetos Site, por medio del sitio de administración de Django, o por medio de la API de Python. Crea un objeto Site para cada sitio/dominio que esté respaldado por este proyecto Django.

4. Define la opción de configuración SITE_ID en cada uno de tus archivos de configuración (*settings*). Este valor debería ser el ID de base de datos del objeto Site para el sitio respaldado por el archivo de configuración.

Las capacidades del framework Sites

Las siguientes secciones describen las cosas que puedes hacer con este *framework*

Rehusar datos en múltiples sitios

Para rehusar los datos en múltiples sitios, como explicamos en el primer escenario, simplemente debes agregarle un campo muchos-a-muchos, ManyToManyField hacia Site en tus modelos.

Por ejemplo:

```
from django.db import models
from django.contrib.sites.models import Site

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    # ...
    sites = models.ManyToManyField(Site)
```

Esa es toda la infraestructura necesaria para asociar el modelo Libro con múltiples sitios en tu base de datos. Con eso en su lugar, puedes rehusar el mismo código de vista para múltiples sitios. Continuando con el modelo Libro del ejemplo, aquí mostramos cómo luciría una vista detalle_libro:

```
from django.conf import settings
from biblioteca.models import Libro

def detalle_libro(request, libro_pk):
    try:
        libro = Libro.objects.get(pk=libro_pk, sites__id=settings.SITE_ID)
    except Libro.DoesNotExist:
        raise Http404

    return render_to_response('biblioteca/detalles_libro.html', {'libro': libro})
```

Esta función de vista es reusable porque comprueba el sitio del artículo dinámicamente, según cuál sea el valor de la opción SITE_ID.

Por ejemplo, digamos que el archivo de configuración de LJWorld.com tiene un SITE_ID asignado a 1, y que el de Lawrence.com lo tiene asignado a 2. Si esta vista es llamada cuando el archivo de configuración de LJWorld.com está activado, entonces la búsqueda de artículos se limita a aquellos en que la lista de sitios incluye LJWorld.com.

Asociación de contenido con un solo sitio

De manera similar, puedes asociar un modelo con el modelo *Site* en una relación muchos-a-uno, usando ForeignKey.

Por ejemplo, si un artículo sólo se permite en un sitio, puedes usar un modelo como este:

```
from django.db import models
from django.contrib.sites.models import Site

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    # ...
    sites = models.ForeignKey(Site)
```

Este tiene los mismos beneficios, como se describe en la última sección.

Obtención del sitio actual desde las vistas

A un nivel más bajo, puedes usar el *framework sites* en tus vistas de Django para hacer cosas particulares, según el sitio en el cual la vista sea llamada. Por ejemplo:

```
from django.conf import settings

def mi_vista(request):
    if settings.SITE_ID == 3:
        # Haz algo.
    else:
        # Haz otra cosa.
```

Por supuesto, es horrible meter el ID en el código del sitio de esa manera. Una forma levemente más limpia de lograr lo mismo, es comprobar el dominio actual del sitio:

```
from django.conf import settings
from django.contrib.sites.models import Site

def mi_vista(request):
    sitio_actual = Site.objects.get(id=settings.SITE_ID)
    if sitio_actual.domain == 'foo.com':
        # Haz algo.
    else:
        # Haz otra cosa.
```

Este fragmento de código usado para obtener el objeto Site según el valor de settings.SITE_ID es tan usado, que el manejador de modelos de Site (Site.objects) tiene un método get_current(). El siguiente ejemplo es equivalente al anterior:

```
from django.contrib.sites.models import Site

def mi_vista(request):
    sitio_actual = Site.objects.get_current()
    if sitio_actual.domain == 'foo.com':
        # Haz algo.
    else:
        # Haz otra cosa.
```

■**Nota:** Observa que en este último ejemplo, no hay necesidad de importar django.conf.settings.

Obtención del dominio actual para ser mostrado

Una forma DRY (acrónimo del inglés *Don't Repeat Yourself*, "no te repitas") de guardar el nombre del sitio y del dominio, como explicamos en "Escenario 2: alojamiento del nombre/dominio de tu sitio en un solo lugar", se logra simplemente haciendo referencia a name y a domain del objeto Site actual. Por ejemplo:

```
from django.contrib.sites.models import Site
from django.core.mail import send_mail

def registrar_para_boletines(request):
    # Verifica los valores del formulario, etc., y suscribe al usuario.
    # ...
    sitio_actual = Site.objects.get_current()
    send_mail('Gracias por suscribirse a %s alertas' % sitio_actual.name,
        'Gracias por suscribirse. Se lo agradecemos.\n\n-El %s equipo.' %
        sitio_actual.name, 'editor@%s' % sitio_actual.domain,
    [user_email])
    # ...
```

Continuando con nuestro ejemplo de LJWorld.com y Lawrence.com, en Lawrence.com el correo electrónico tiene como sujeto la línea "Gracias por suscribirse a las alertas de lawrence.com". En LJWorld.com, en cambio, el sujeto es "Gracias por suscribirse a las alertas de LJWorld.com". Este comportamiento específico para cada sitio, también se aplica al cuerpo del correo electrónico.

Una forma aún más flexible (aunque un poco más pesada) de hacer lo mismo, es usando el sistema de plantillas de Django. Asumiendo que Lawrence.com y LJWorld.com tienen distintos directorios de plantillas (TEMPLATE_DIRS), puedes simplemente delegarlo al sistema de plantillas así:

```
from django.core.mail import send_mail
from django.template import loader, Context

def registrar_para_boletines(request):
    # Verifica los valores del formulario, etc., y suscribe al usuario.
    # ...
    asunto = loader.get_template('alertas/asunto.txt').render(Context({}))
    mensaje = loader.get_template('alertas/mensaje.txt').render(Context({}))
    send_mail(asunto, mensaje, 'do-not-reply@example.com', [user_email])
    # ...
```

En este caso, debes crear las plantillas asunto.txt y mensaje.txt en ambos directorios de plantillas, el de LJWorld.com y el de Lawrence.com . Como mencionamos anteriormente, eso te da más flexibilidad, pero también es más complejo.

Una buena idea es explotar los objetos Site lo más posible, para que no haya complejidad y redundancia innecesaria.

Obtención del dominio actual para las URLs completas

La convención de Django de usar get_absolute_url() para obtener las URLs de los objetos sin el dominio, está muy bien. Pero en algunos casos puedes querer mostrar la URL completa – con http:// y el dominio, y todo – para un objeto. Para hacerlo, puedes usar el *framework sites*. Este es un ejemplo:

```
>>> from django.contrib.sites.models import Site
>>> from biblioteca.models import Libro
>>> obj = Libro.objects.get(id=3)
>>> obj.get_absolute_url()
'/detalle/libro/3/'
>>> Site.objects.get_current().domain
'localhost:9000'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://localhost:9000/detalle/libro/3/'
```

El manejador CurrentSiteManager

Si el modelo Site juegan roles importantes en tu aplicación, considera el uso del útil manejador llamado: CurrentSiteManager en tu modelo (o modelos). Es un administrador de modelos (consulta el Apéndice B) que filtra automáticamente sus consultas para incluir sólo los objetos asociados al Site actual.

Usa CurrentSiteManager agregándolo a tu modelo explícitamente. Por ejemplo:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    # Agregamos una relacion foranea a "Sites"
    sites = models.ForeignKey(Site)
    # El manejador por defecto.
    objects = models.Manager()
    # Agregamos el manejador para "Sites"
    mi_sitio = CurrentSiteManager()
```

Con este modelo, Libro.objects.all() retorna todos los objetos Libro de la base de datos, pero Libro.mi_sitio.all() retorna sólo los objetos Libro asociados con el sitio actual, de acuerdo a la opción de configuración SITE_ID.

En otras palabras, estas dos sentencias son equivalentes:

```
Libro.objects.filter(site=settings.SITE_ID)

Libro.mi_sitio.all()
```

¿Cómo supo CurrentSiteManager cuál campo de Libro era el Site? Por defecto busca un campo llamado site. Si tu modelo tiene un campo ForeignKey o un campo ManyToManyField llamado de otra forma que site, debes pasarlo explícitamente como el parámetro para CurrentSiteManager.

El modelo a continuación, tiene un campo llamado publicado_en, como lo demuestra el siguiente ejemplo:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    # Agrega una relacion foranea usando otro nombre en lugar de "sites"
    publicado_en = models.ForeignKey(Site)
    # El manejador por defecto.
    objects = models.Manager()
    # Agrega  al manejador el nombre del campo: "Sites"
    mi_sitio = CurrentSiteManager('publicado_en')
```

Si intentas usar CurrentSiteManager y pasarle un nombre de campo que no existe, Django lanzará un error del tipo: ValueError.

Nota: Probablemente querrás tener un Manager normal (no específico al sitio) en tu modelo, incluso si usas CurrentSiteManager. Como se explica en el Apéndice B, si defines un *manager* manualmente, Django no creará automáticamente el *manager* objects = models.Manager().

Además, algunas partes de Django – el sitio de administración y las vistas genéricas – usan el *manager* que haya sido definido *primero* en el modelo. Así que si quieres que el sitio de administración tenga acceso a todos los objetos (no sólo a los específicos al sitio actual), pon un objects = models.Manager() en tu modelo, antes de definir CurrentSiteManager.

Si utilizas a menudo esta configuración, en lugar de hacer esto en tus vistas:

```
from django.contrib.sites.models import Site

def mi_vista(request):
    site = Site.objects.get_current()
    ...
```

Existe una manera simple de evitar repeticiones. Usando Middleware, solo agrega django.contrib.sites.middleware.CurrentSiteMiddleware a la variable MIDDLEWARE_CLASSES. El middleware se encarga de configurar estos atributos en cada petición de un objeto, por lo que puedes usar request.site para obtener el sitio actual.

Cómo utilizar Django el framework Sites

Si bien no es necesario que uses el *framework sites*, es extremadamente recomendado, porque Django toma ventaja de ello en algunos lugares. Incluso si tu instalación de Django está alimentando a un solo sitio, deberías tomarte unos segundos para crear el objeto *site* con tu domain y name, y apuntar su ID en tu opción de configuración SITE_ID.

Este es el uso que hace Django del *framework sites*:

- En el *framework redirects* (consulta la sección "Redirects" más adelante), cada objeto *redirect* está asociado con un sitio en particular. Cuando Django busca un *redirect*, toma en cuenta el SITE_ID actual.

- En el *framework flatpages* (consulta la sección "Flatpages" más adelante), cada página es asociada con un sitio en particular. Cuando una página es creada, tú especificas su site, y el *middleware* de *flatpage* chequea el SITE_ID actual cuando se traen páginas para ser mostradas.

- En el *framework syndication* (consulta el *capítulo 13*.), las plantillas para title y description tienen acceso automático a la variable {{ site }}, que es el objeto Site que representa al sitio actual. Además, la conexión para proporcionar las URLs de los elementos usan el domain desde el objeto Site actual si no especificas un nombre de dominio.

- En el *framework authentication* (consulta el *capítulo 14*.), la vista django.contrib.auth.views.login le pasa el nombre del Site actual a la plantilla como {{ site_name }}.

Flatpages

A menudo tendrás una aplicación Web impulsada por una bases de datos ya funcionando, pero necesitarás agregar un par de páginas estáticas, tales como una página *Acerca de* o una página de Política de Privacidad. Sería posible usar un servidor Web estándar como por ejemplo Apache para servir esos archivos como archivos HTML planos, pero eso introduce un nivel extra de complejidad en tu aplicación, porque entonces tienes que preocuparte de la configuración de Apache, tienes que preparar el acceso para que tu equipo pueda editar esos archivos, y no puedes sacar provecho del sistema de plantillas de Django para darle estilo a las páginas.

La solución a este problema es la aplicación flatpages de Django, la cual reside en el paquete django.contrib.flatpages. Esta aplicación te permite manejar esas páginas aisladas mediante el sitio de administración de Django, y te permite especificar plantillas para las mismas usando el sistema de plantillas de Django. Detrás de escena usa modelos Django, lo que significa que almacena las páginas en una base de datos, de la misma manera que el resto de tus datos, y puedes acceder a las flatpages con la API de bases de datos estándar de Django.

Las flatpages son identificadas por su URL y su sitio. Cuando creas una flatpage, especificas con cual URL está asociada, junto con en cuál(es) sitio(s) está (para más información acerca de sitios, consulta la sección "Sites").

Usar Flatpages

Para instalar la aplicación flatpages, sigue estos pasos:

1. Agrega 'django.contrib.flatpages' a tu INSTALLED_APPS.
   django.contrib.flatpages depende de django.contrib.sites, asi que asegúrate de que ambos paquetes se encuentren en INSTALLED_APPS.

2. Agrega 'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware' a tu variable de configuración MIDDLEWARE_CLASSES.

3. Ejecuta el comando python manage.py migrate para instalar las dos tablas necesarias en tu base de datos.

La aplicación flatpages crea dos tablas en tu base de datos: django_flatpage y django_flatpage_sites. django_flatpage simplemente mantiene una correspondencia entre URLs y títulos más contenidos de texto. django_flatpage_sites es una tabla muchos a muchos que asocia una flatpage con uno o más sitios.

La aplicación incluye un único modelo FlatPage, definido en django/contrib/flatpages/models.py. El mismo se ve así:

```
from django.db import models
from django.contrib.sites.models import Site

class FlatPage(models.Model):
    url = models.CharField(maxlength=100)
    title = models.CharField(maxlength=200)
    content = models.TextField()
    enable_comments = models.BooleanField()
    template_name = models.CharField(maxlength=70, blank=True)
    registration_required = models.BooleanField()
    sites = models.ManyToManyField(Site)
```

Examinemos cada uno de los campos:

- **url:** La URL donde reside esta flatpage, excluyendo el nombre del dominio pero incluyendo la barra (/) inicial (por ej. /about/contact/).

- **title:** El título de la flatpage. El framework no usa esto para nada en especial. Es tu responsabilidad visualizarlo en tu plantilla.

- **content:** El contenido de la flatpage (por ej. el HTML de la página). El framework no usa esto para nada en especial. Es tu responsabilidad visualizarlo en tu plantilla.

- **enable_comments**: Indica si deben activarse los comentarios e esta flatpage. El framework no usa esto para nada en especial. Puedes comprobar este valor en tu plantilla y mostrar un formulario de comentario si es necesario.

- **template_name:** El nombre de la plantilla a usarse para renderizar esta flatpage. Es opcional; si no se indica o si esta plantilla no existe, el framework usará la plantilla flatpages/default.html.

- **registration_required:** Indica si se requerirá registro para ver esta flatpage. Esto se integra con el framework de autenticación/usuarios de Django, el cual se trata en él capítulo 12.

- **sites:** Los sitios en los cuales reside esta flatpage. Esto se integra con el framework sites de Django, el cual se trata en la sección "Sites" en este capítulo.

Puedes crear flatpages ya sea a través de la interfaz de administración de Django o a través de la API de base de datos de Django. Para más información, examina la sección "*Agregando, modificando y eliminando flatpages*".

Una vez que has creado flatpages, FlatpageFallbackMiddleware se encarga de todo el trabajo. Cada vez que cualquier aplicación Django lanza un error, este middleware verifica como último recurso la base de datos de flatpages en búsqueda de la URL que se ha requerido.

Específicamente busca una flatpage con la URL en cuestión y con un identificador de sitio que coincida con la variable de configuración SITE_ID.

Si encuentra una coincidencia, carga la plantilla de la flatpage, o flatpages/default.html si la flatpage no ha especificado una plantilla personalizada. Le pasa a dicha plantilla una única variable de contexto: flatpage, la cual es el objeto flatpage. Usa RequestContext para renderizar la plantilla.

Si FlatpageFallbackMiddleware no encuentra una coincidencia, el proceso de la petición continúa normalmente.

Nota: Este middleware sólo se activa para errores 404 (página no encontrada) – no para errores 500 (error en servidor) u otras respuestas de error. Nota también que el orden de MIDDLEWARE_CLASSES es relevante. Generalmente, puedes colocar el FlatpageFallbackMiddleware cerca o en el final de la lista, debido a que se trata de una opción de último recurso.

Agregar, modificar y eliminar flatpages

Puedes agregar, cambiar y/o eliminar paginas estáticas o flatpages de dos maneras:

1. **A través de la interfaz de administración**

Si has activado la interfaz automática de administración de Django, deberías ver una sección "Flatpages" en la página de índice de la aplicación admin. Edita las flatpages como lo harías con cualquier otro objeto en el sistema.

2. **A través de de la API Python**

Como ya se describió, las flatpages se representan mediante un modelo Django estándar que reside en django/contrib/flatpages/models.py. Por lo tanto puede acceder a objetos flatpage mediante la API de base de datos Django, por ejemplo:

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from django.contrib.sites.models import Site
>>> fp = FlatPage(
... url='/acerca/',
... title='Acerca de ',
... content='<p>Acerca de este sitio...</p>',
... enable_comments=False,
... template_name='',
... registration_required=False,
... )
>>> fp.save()
>>> fp.sites.add(Site.objects.get(id=1))
>>> FlatPage.objects.get(url='/acerca/')
<FlatPage: /acerca/ -- Acerca de>
```

Usar plantillas de flatpages

Por omisión, las flatpages son renderizadas vía la plantilla flatpages/default.html, pero puedes cambiar eso para cualquier flatpage con el campo template_name en el objeto FlatPage.

Es tu responsabilidad el crear la plantilla flatpages/default.html. En tu directorio de plantillas, crea un directorio flatpages que contenga un archivo default.html.

A las plantillas de flatpages se les pasa una única variable de contexto: flatpage, la cual es el objeto flatpage.

Este es un ejemplo de una plantilla flatpages/default.html:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
   "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
   <title>{{ flatpage.title }}</title>
</head>
<body>
    {{ flatpage.content }}
</body>
</html>
```

Redirects

El framework redirects de Django te permite administrar las redirecciones con facilidad almacenándolos en una base de datos y tratándolos como cualquier otro objeto modelo de Django. Por ejemplo puedes usar el framework redirects para indicarle a Django “Redirecciona cualquier petición de /musica/ a /seccion/artista/musica/.”. Esto es útil cuando necesitas cambiar las cosas de lugar en tu sitio; los desarrolladores Web deberían hacer lo que esté en sus manos para evitar los enlaces rotos.

Usando el framework redirects

Para instalar la aplicación redirects, sigue estos pasos:

- Agrega 'django.contrib.redirects' a tu INSTALLED_APPS.
- Agrega 'django.contrib.redirects.middleware.RedirectFallbackMiddleware' a tu variable de configuración MIDDLEWARE_CLASSES.
- Ejecuta el comando `python manage.py migrate` para instalar la única tabla necesaria a tu base de datos.

  `manage.py migrate` crea una tabla django_redirect en tu base de datos. Esta se trata sencillamente de una tabla de búsqueda con campos site_id, old_path y new_path.

Puedes crear redirecciones tanto a través de la interfaz de administración como a través de la API de base de datos de Django. Para más información puedes leer la sección “Agregar, modificar y eliminar redirecciones”.

Una vez que has creado redirecciones, la clase RedirectFallbackMiddleware se encarga de todo el trabajo. Cada vez que cualquier aplicación Django lanza un error 404, este middleware verifica como último recurso la base de datos de redirects en búsqueda de la URL que se ha requerido. Específicamente busca un redirect con el old_path provisto y con un identificador de sitio que coincida con la variable de configuración SITE_ID. (para más información acerca de SITE_ID y el framework sites, consulta la sección "Sites").

Luego entonces realiza los siguientes pasos:

- Si encuentra una coincidencia y new_path no está vacío, redirecciona la petición a new_path.
- Si encuentra una coincidencia y new_path está vació, envía una cabecera HTTP 410 ("Ausente") y una respuesta vacía (sin contenido).
- Si no encuentra una coincidencia, el procesamiento de la petición continúa normalmente.

El middleware sólo se activa ante errores 404 – no en errores 500 o respuestas con otros códigos de estado.

■**Nota:** que el orden de MIDDLEWARE_CLASSES es relevante. Generalmente puedes colocar RedirectFallbackMiddleware cerca del final de la lista, debido a que se trata de una opción de último recurso.

Si usas los middlewares redirect y flatpages, analiza cual de los dos (redirect o flatpages) desearías sea ejecutado primero. Sugerimos configurar flatpages antes que redirects (o sea colocar el middleware flatpages antes que el middleware redirects) pero tú podrías decidir lo contrario.

Agregar, modificar y eliminar redirecciones

Puedes agregar, modificar y eliminar redirecciones de dos maneras:

1. **A través de la interfaz de administración**

   Si has activado la interfaz automática de administración de Django, deberías ver una sección "Redirects" en la página de índice de la aplicación admin. Edita las redirecciones como lo harías con cualquier otro objeto en el sistema.

2. **A través de la API Python**

   Las redirecciones se representan mediante un modelo estándar Django que reside en django/contrib/redirects/models.py. Por lo tanto puedes acceder a los objetos redirect vía la API de base de datos de Django, por ejemplo:

```
>>> from django.contrib.redirects.models import Redirect
>>> from django.contrib.sites.models import Site
>>> red = Redirect(
... site=Site.objects.get(id=1),
... old_path='/musica/',
... new_path='/seccion/artista/musica/',
```

```
... )
>>> red.save()
>>> Redirect.objects.get(old_path='/musica/')
<Redirect: /music/ ---> /secccion/artista/musica/>
```

Protección contra CSRF

El paquete django.contrib.csrf provee protección contra *Cross-site request forgery* (CSRF) (falsificación de peticiones inter-sitio).

CSRF, también conocido como *"session riding"* (montado de sesiones) es un exploit de seguridad en sitios Web. Se presenta cuando un sitio Web malicioso induce a un usuario a cargar sin saberlo una URL desde un sitio al cual dicho usuario ya se ha autenticado, por lo tanto saca ventaja de su estado autenticado. Inicialmente esto puede ser un poco difícil de entender así que en esta sección recorreremos un par de ejemplos.

Un ejemplo simple de CSRF

Supongamos que posees una cuenta de *correo electronico* en example.com. Este sitio proveedor de *correo* tiene un botón *Log Out* que apunta a la URL example.com/logout – esta es, la única acción que necesitas realizar para desconectarte (*log out*) es visitar la página example.com/logout.

Un sitio malicioso puede obligarte a visitar la URL example.com/logout incluyendo esa URL como un <iframe> oculto en su propia página maliciosa. De manera que si estás conectado (*logged in*) a tu cuenta de *correo* del sitio example.com y visitas la página maliciosa, el hecho de visitar la misma te desconectará de example.com.

Claramente, ser desconectado de un sitio de *correo* contra tu voluntad no es un incidente de seguridad aterrorizante, pero este tipo de exploit puede sucederle a *cualquier* sitio que "confía" en sus usuarios, tales como un sitio de un banco o un sitio de comercio electrónico.

Un ejemplo más complejo de CSRF

En el ejemplo anterior, el sitio example.com tenía parte de la culpa debido a que permitía que se pudiera solicitar un cambio de estado (la desconexión del sitio) mediante el método HTTP GET.

Es una práctica mucho mejor el requerir el uso de un POST HTTP para cada petición que cambie el estado en el servidor. Pero aun los sitios Web que requieren el uso de POST para acciones que signifiquen cambios de estado son vulnerables a CSRF.

Supongamos que example.com ha mejorado su funcionalidad de desconexión de manera que "Log Out" es ahora un botón de un <form> que es enviado vía un POST a la URL example.com/logout. Adicionalmente, el <form> de desconexión incluye un campo oculto:

<input type="hidden" name="confirm" value="true" />

Esto asegura que un simple POST a la URL example.com/logout no desconectará a un usuario; para que los usuarios puedan desconectarse, deberán enviar una petición a example.com/logout usando POST *y* enviar la variable POST confirm con el valor 'true'.

Bueno, aun con dichas medidas extra de seguridad, este esquema también puede ser atacado mediante CSRF – la página maliciosa sólo necesita hacer un poquito más de trabajo. Los atacantes pueden crear un formulario completo que envíe su petición a tu sitio, ocultar el mismo en un <iframe> invisible y luego usar JavaScript para enviar dicho formulario en forma automática.

Previniendo CSRF

Entonces, ¿Cómo puede tu sitio defenderse de este exploit?. El primer paso es asegurarse que todas las peticiones GET no posean efectos colaterales. De esa forma, si un sitio malicioso incluye una de tus páginas como un <iframe>, esto no tendrá un efecto negativo.

Esto nos deja con las peticiones POST. El segundo paso es dotar a cada <form> que se enviará vía POST un campo oculto cuyo valor sea secreto y sea generado en base al identificador de sesión del usuario. Entonces luego, cuando se esté realizando el procesamiento del formulario en el servidor, comprobar dicho campo secreto y generar un error si dicha comprobación no es exitosa.

Esto es precisamente lo que hace la capa de prevención de CSRF de Django, tal como se explica en la siguiente sección.

Usar el middleware CSRF

El paquete django.contrib.csrf contiene sólo un módulo: middleware.py. Este módulo contiene una clase middleware Django: CsrfMiddleware la cual implementa la protección contra CSRF, por defecto.

Para activar esta protección en tiempos de ejecucucion, solo agrega 'django.contrib.csrf.middleware.CsrfMiddleware' a la variable de configuración MIDDLEWARE_CLASSES en tu archivo de configuración. Este middleware necesita procesar la respuesta *después* de SessionMiddleware, así que CsrfMiddleware debe aparecer *antes* que SessionMiddleware en la lista (esto es debido que el middleware de respuesta es procesado de atrás hacia adelante). Por otra parte, debe procesar la respuesta antes que la misma sea comprimida o alterada de alguna otra forma, de manera que CsrfMiddleware debe aparecer después de GZipMiddleware. Una vez que has agregado eso a tu MIDDLEWARE_CLASSES ya estás listo para usarlo, tanto en plantillas como en vistas.

En cualquier plantilla que use un formulario POST, usa la etiqueta csrf_token dentro de los elementos <form> si el formulario es para una URL interna. Por ejemplo:

```
<form action="." method="post">{% csrf_token %}
```

Esto no se debe de hacer en formularios POST que apunten a URLs externas, ya que esto causaría que el CSRF sea escapado, conduciéndonos a problemas de vulnerabilidad.

En las vistas, solo asegúrate de que el procesador de contexto: django.core.context_processors.csrf sea usado. Usualmente puedes hacerlo de dos formas:

1. Usando RequestContext, el cual siempre usa django.core.context_processors.csrf (no importando la configuración de TEMPLATE_CONTEXT_PROCESSORS). Si estas usando una vista genérica o una aplicación del paquete contrib, estas a salvo, ya que estas aplicaciones ya incluyen RequestContext por defecto.

2. Manualmente importa y usa el procesador generado por el token CSRF y agrégalo en el contexto de la plantilla, así:

```
from django.core.context_processors import csrf
from django.shortcuts import render_to_response

def mi_vista(request):
    c = {}
    c.update(csrf(request))
    # ... el código de la vista aquí.
    return render_to_response("mi plantilla.html", c)
```

Finalmente puedes usar un decorador que envuelva tu vista y que tome en cuenta estos pasos por ti.

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def mi_vista(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

☠**Advertencia:** El uso del decorador por sí mismo no es muy recomendable, ya que si olvidas utilizarlo, tendrás un gran agujero de seguridad.

En el caso en el que estés interesado, así es como trabaja CsrfMiddleware. Realizando estas dos cosas:

1. Modifica las respuestas salientes a peticiones agregando un campo de formulario oculto a todos los formularios POST, con el nombre csrfmiddlewaretoken y un valor que es un *hash* del identificador de sesión más una clave secreta. El middleware *no* modifica la respuesta si no existe un identificador de sesión, de manera que el costo en rendimiento es despreciable para peticiones que no usan sesiones.

2. Para todas las peticiones POST que porten la cookie de sesión, comprueba que csrfmiddlewaretoken esté presente y tenga un valor correcto. Si no cumple estas condiciones, el usuario recibirá un error HTTP 403. El contenido de la página de error es el mensaje "Cross Site Request Forgery detected. Request aborted."

Esto asegura que solamente se puedan usar formularios que se hayan originado en tu sitio Web para enviar datos vía POST al mismo.

Este middleware deliberadamente trabaja solamente sobre peticiones HTTP POST (y sus correspondientes formularios POST). Como ya hemos explicado, las peticiones GET nunca deberían tener efectos colaterales; es tu responsabilidad asegurar eso.

Las peticiones POST que no estén acompañadas de una cookie de sesión no son protegidas simplemente porque no tiene sentido protegerlas, un sitio Web malicioso podría de todas formas generar ese tipo de peticiones.

Para evitar alterar peticiones no HTML, el middleware revisa la cabecera Content-Type de la respuesta antes de modificarla. Sólo modifica las páginas que son servidas como text/html o application/xml+xhtml.

Limitaciones del middleware CSRF

CsrfMiddleware necesita el framework de sesiones de Django para poder funcionar. (Revisa el *capítulo 14* para obtener más información sobre sesiones). Si estás usando un framework de sesiones o autenticación personalizado que maneja en forma manual las cookies de sesión, este middleware no te será de ayuda.

Si tu aplicación crea páginas HTML y formularios con algún método inusual (por ej. si envía fragmentos de HTML en sentencias JavaScript document.write), podrías estar salteándote el filtro que agrega el campo oculto al formulario. De presentarse esta situación, el envío del formulario fallará siempre. (Esto sucede porque CsrfMiddleware usa una expresión regular para agregar el campo csrfmiddlewaretoken a tu HTML antes de que la página sea enviada al cliente, y la expresión regular a veces no puede manejar código HTML muy extravagante). Si sospechas que esto podría estar sucediendo, sólo examina el código en tu navegador Web para ver si es que csrfmiddlewaretoken ha sido insertado en tu <form>.

Para más información y ejemplos sobre CSRF, visita
http://en.wikipedia.org/wiki/CSRF.

Humanizando Datos

El paquete contrib.humanize es una aplicación que aloja un conjunto de filtros de plantilla útiles a la hora de agregar un "toque humano" a los datos. Para activar esos filtros, agrega 'django.contrib.humanize' a tu variable de configuración INSTALLED_APPS.

Una vez que has hecho eso, carga las etiquetas con {% load humanize %} en una plantilla, y tendrás acceso a los filtros que se describen en las siguientes secciones.

apnumber

Para números entre 1 y 9, este filtro retorna la representación textual del número. Caso contrario retorna el numeral. Esto cumple con el estilo Associated Press.

Ejemplos:

- 1 se convierte en uno.
- 2 se convierte en dos.
- 0 se convierte en 10.

Puedes pasarle ya sea un entero o una representación en cadena de un entero.

intcomma

Este filtro convierte un entero a una cadena conteniendo comas cada tres dígitos.

Ejemplos:

- 4500 se convierte en 4,500.
- 45000 se convierte en 45,000.

- 450000 se convierte en 450,000.
- 4500000 se convierte en 4,500,000.

Puedes pasarle ya sea un entero o una representación en cadena de un entero.

intword

Este filtro convierte un entero grande a una representación amigable en texto. Funciona mejor con números mayores a un millón.
Ejemplos:

- 1000000 se convierte en 1.0 millón.
- 1200000 se convierte en 1.2 millón.
- Se admiten valores hasta de 10^100 (Googol).

Puedes pasarle ya sea un entero o una representación en cadena de un entero.

naturalday

Este filtro se usa para mostrar fechas por ejemplo hoy, mañana o ayer. Tomando como valor la fecha del día de hoy.

Ejemplos (cuando la etiqueta 'now' es 17 Feb 2007):

- 16 Feb 2007 se convierte en aller.
- 17 Feb 2007 se convierte en hoy.
- 18 Feb 2007 se convierte en mañana.

Cualquier otro día se formatea según el argumento dado o configurado en DATE_FORMAT si no se da ningún argumento.
Argumentos: Una fecha formateada como una cadena.

naturaltime

Este filtro se usa para mostrar valores de tiempo, retorna una cadena que representa cuantos segundos, minutos y horas han pasado –recurre al formato de la etiqueta timesince si el valor tiene más de un día. En este caso retorna el valor usando una frase.
Ejemplo (cuando la etiqueta 'now' es 17 Feb 2007 16:30:00):

- 17 Feb 2007 16:30:00 se convierte en ahora.
- 17 Feb 2007 16:29:31 se convierte en hace 29
- 17 Feb 2007 16:29:00 se convierte en hace un minuto.
- 17 Feb 2007 16:25:35 se convierte en hace 4 minutos.
- 17 Feb 2007 15:30:29 se convierte en Hace 59 minutos.

ordinal

Este filtro convierte un entero a una cadena cuyo valor es su ordinal.
Ejemplos:

- 1 se convierte en 1st.
- 2 se convierte en 2nd.

- 3 se convierte en 3rd.

Puedes pasarle ya sea un entero o una representación en cadena de un entero.

¿Qué sigue?

Muchos de estos frameworks contribuidos (CSRF, el sistema de autenticación, etc.) hacen su magia proveyendo una pieza de middleware. El middleware es esencialmente código que se ejecuta antes y/o después de cada petición y puede modificar cada petición y respuesta a voluntad. A continuación trataremos el middleware incluido con Django y explicaremos cómo puedes crear tu propio middleware.

CAPÍTULO 17

Middleware

En ocasiones, necesitarás ejecutar una pieza de código en todas las peticiones que maneja Django. Éste código puede necesitar modificar la petición antes de que la vista se encargue de ella, puede necesitar registrar información sobre la petición para propósitos de debugging, y así sucesivamente.

Tu puedes hacer esto con el *framework middleware* de Django, que es un conjunto de acoples dentro del procesamiento de petición/respuesta de Django. Es un sistema de "plug-in" liviano y de bajo nivel capaz de alterar de forma global tanto la entrada como la salida de Django.

Cada componente middleware es responsable de hacer alguna función específica. Si estás leyendo este libro de forma lineal (disculpen, posmodernistas), has visto middleware varias veces ya:

Todas las herramientas de usuario y sesión que vimos en el *capítulo 14* son posibles gracias a unas pequeñas piezas de middleware (más específicamente, el middleware hace que request.session y request.user estén disponibles para ti en las vistas.

La cache global del sitio discutida en el *capítulo 15* es solo una pieza de middleware que desvía la llamada a tu función de vista si la respuesta para esa vista ya fue almacenada en la cache.

Todas las aplicaciones del paquete contrib como flatpages, redirects, y csrf del *capítulo 16* hacen su magia a través de componentes middleware.

En este capítulo nos sumergiremos en las profundidades del middleware y conoceremos exactamente cómo funciona, y te explicaremos cómo puedes escribir tu propio middleware.

¿Qué es el Middleware?

Un componente middleware es simplemente una clase Python que se ajusta a una cierta API. Antes de entrar en los aspectos formales de los que es esa API, miremos un ejemplo muy sencillo.

Sitios de tráfico alto a menudo necesitan implementar Django detrás de un proxy de balanceo de carga (mira él *capítulo 12*). Esto puede causar unas pequeñas complicaciones, una de las cuales es que la IP remota de cada petición (request.META["REMOTE_IP"]) será la del balanceador de carga, no la IP real que realiza la petición. Los balanceadores de carga manejan esto estableciendo una cabecera especial, X-Forwarded-For, con el valor real de la dirección IP que realiza la petición.

Así que aquí está una pequeña parte de middleware que le permite a los sitios que se ejecutan detrás de un proxy ver la dirección IP correcta en request.META["REMOTE_ADDR"]:

```
class SetRemoteAddrFromForwardedFor(object):
    def  process_request(self, request):
    try
        real_ip = request.META['HTTP_X_FORWARDED_FOR']
    except KeyError:
        pass
    else:
        # HTTP_X_FORWARDED_FOR puede ser una lista de IP's separadas
        # por comas.  Toma la primera IP que encuentra.
        real_ip = real_ip.split(",")[0]
```

(Nota: Aunque las cabeceras HTTP son llamadas X-Forwarded-For, Django hace que estén disponibles como request.META['HTTP_X_FORWARDED_FOR']. Con la excepción de content-length y content-type, cualquier cabecera HTTP en la petición es convertida en una clave request.META convirtiendo todos los caracteres a mayúsculas, remplazando cualquier guion con guiones bajos y agregando el prefijo HTTP_ al nombre.

Si el middleware está instalado (mira la siguiente sección), el valor de de todas las peticiones X-Forwarded-For será automáticamente insertado en request.META['REMOTE_ADDR']. Esto significa que tus aplicaciones Django no necesitan conocer si están detrás de un proxy de balanceo de carga o no, pueden simplemente acceder a request.META['REMOTE_ADDR'], y eso funcionará si se usa un proxy o no.

De hecho, es una necesidad tan común, que esta pieza de middleware ya viene incorporada en Django. Está ubicada en django.middleware.http, y puedes leer más sobre ella en la siguiente sección.

Instalación de Middleware

Si has leído este libro completamente hasta aquí, ya has visto varios ejemplos de instalación de middleware; muchos de los ejemplos en los capítulos previos han requerido cierto middleware. Para completar, a continuación se muestra la manera de instalar middleware.

Para activar algún componente del middleware, solo agrégalo a la tupla MIDDLEWARE_CLASSES en tu archivo de configuración. En MIDDLEWARE_CLASSES, cada componente middleware se representa con una cadena: la ruta Python completa al nombre de la clase middleware. Por ejemplo, aquí se muestra la tupla MIDDLEWARE_CLASSES por omisión creada por django-admin.py startproject:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
)
```

Una instalación Django no requiere ningún middleware – La tupla MIDDLEWARE_CLASSES puede estar vacía, si tu quieres, pero te recomendamos que actives CommonMiddleware, la cual explicaremos en breve.

El orden es importante. En las fases de petición y vista, Django aplica el middleware en el orden que figura en MIDDLEWARE_CLASSES, y en las fases de respuesta y excepción, Django aplica el middleware en el orden inverso. Es decir, Django trata MIDDLEWARE_CLASSES como una especie de "wrapper" alrededor de la función de vista: en la petición recorre hacia abajo la lista hasta la vista, y en la respuesta la recorre hacia arriba.

Mira la siguiente figura para un repaso de las fases y el orden que sigue el middleware en una petición.

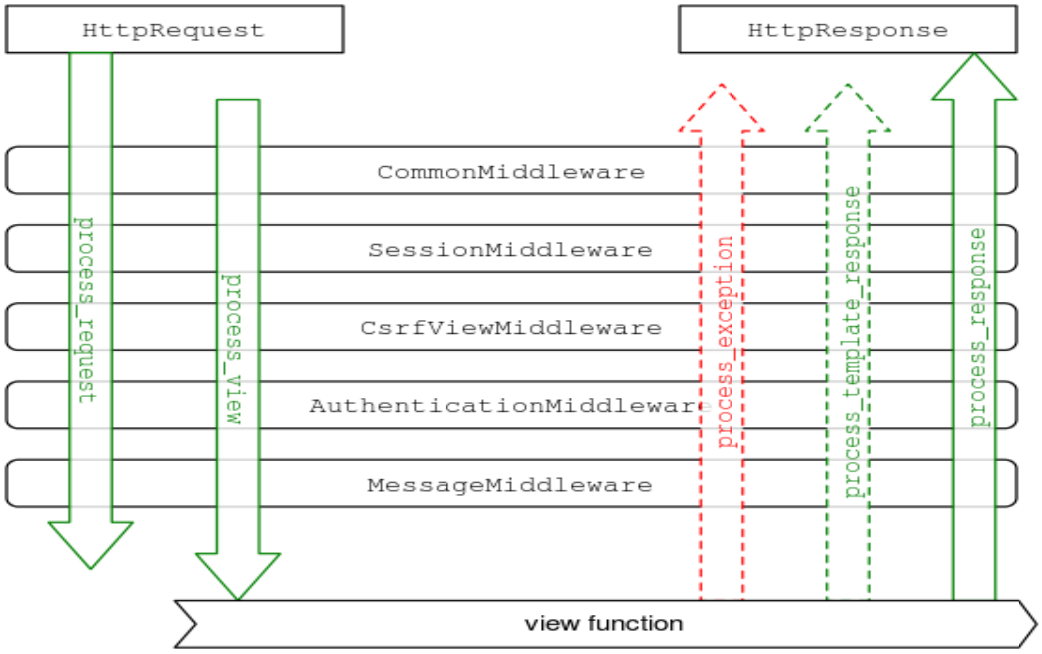


Figura 17-1. Orden de Middleware en Django.

Como puedes observar, durante la fase de petición (request), antes de llamar a la vista, Django aplica el Middleware en el orden definido en la tupla MIDDLEWARE_CLASSES. De arriba hacia abajo. Mediante dos ganchos disponibles:

1. process_request()
2. process_view()

Durante la fase de respuesta, después de llamar a la vista, el middleware es aplicado en orden inverso de abajo hacia arriba, Usando tres ganchos disponibles:

- process_exception() (únicamente si la vista lanza una excepción)
- process_template_response() (únicamente para la respuesta de la plantilla)
- process_response()

Puedes pensar en este proceso, como si fuera una cebolla, donde cada clase del middleware es una capa que envuelve la vista.

Métodos de un Middleware

Ahora que sabes qué es un middleware y cómo instalarlo, echemos un vistazo a todos los métodos disponibles que las clases middleware pueden definir.

Initializar: __init__(self)

Utiliza __init__() para realizar una configuración a nivel de sistema de una determinada clase middleware.

Por razones de rendimiento, cada clase middleware activada es instanciada sólo *una vez* por proceso servidor. Esto significa que __init__() es llamada sólo una vez – al iniciar el servidor, no para peticiones individuales.

Una razón común para implementar un método __init__() es para verificar si el middleware es en realidad necesario. Si __init__() emite django.core.exceptions.MiddlewareNotUsed, entonces Django removerá el middleware de la pila de middleware. Tú podrías usar esta característica para verificar si existe una pieza de software que la clase middleware requiere, o verificar si el servidor esta ejecutándose en modo *debug*, o cualquier otra situación similar.

Si una clase middleware define un método __init__(), éste no debe tomar argumentos más allá del estándar self.

Pre-procesador de petición: process_request(self, request)

Éste método es llamado tan pronto como la petición ha sido recibida – antes de que Django haya analizado sintácticamente la URL para determinar cuál vista ejecutar. Si se le pasa el objeto HttpRequest, el cual puedes modificar a tu voluntad, process_request() debe retornar ya sea None o un objeto HttpResponse.

Si devuelve None, Django continuará procesando esta petición, ejecutando cualquier otro middleware y la vista apropiada.

Si devuelve un objeto HttpResponse, Django no se encargará de llamar a *cualquier* otro middleware (de ningún tipo) o a la vista apropiada. Django inmediatamente devolverá ése objeto HttpResponse.

Pre-procesador de vista: process_view(self, request, view, args, kwargs)

Éste método es llamado después de la llamada al pre-procesador de petición y después de que Django haya determinado qué vista ejecutar, pero antes de que ésa vista sea realmente ejecutada.

Los argumentos que se pasan a esta vista son mostrados en la Tabla 15-1.

Argumento	Explicación
request	El objeto HttpRequest.
view	La función Python que Django llamará para manejar esta petición. Este es en realidad el objeto función en sí, no el nombre de la función como string.
args	La lista de argumentos posicionales que serán pasados a la vista, no incluye el argumento request (el cual es siempre el primer argumento de una vista).
kwargs	El diccionario de palabras clave argumento que será pasado a la vista.

Tabla 17-1. Argumentos que se pasan a process_view()

Así como el método process_request(), process_view() debe retornar ya sea None o un objeto HttpResponse.

- Si devuelve None, Django continuará procesando esta petición, ejecutando cualquier otro middleware y la vista apropiada.

- Si devuelve un objeto HttpResponse, Django no se encargará de llamar a *cualquier* otro middleware (de ningún tipo) o a la vista apropiada. Django inmediatamente devolverá ése objeto HttpResponse.

Pos-procesador de respuesta: process_response(self, request, response)

Éste método es llamado después de que la función de vista es llamada y la respuesta generada.

Aquí, el procesador puede modificar el contenido de una respuesta; un caso de uso obvio es la compresión de contenido, como por ejemplo la compresión con gzip del HTML de la respuesta.

Los parámetros deben ser bastante auto-explicativos: request es el objeto petición, y response es el objeto respuesta retornados por la vista.

A diferencia de los pre-procesadores de petición y vista, los cuales pueden retornar None, process_response() *debe* retornar un objeto HttpResponse. Esa respuesta puede ser la respuesta original pasada a la función (posiblemente modificada) o una totalmente nueva.

Pos-procesador de excepción: process_exception(self, request, excepción)

Éste método es llamado sólo si ocurre algún error y la vista emite una excepción sin capturar.

Puedes usar este método para enviar notificaciones de error, volcar información postmórtem a un registro, o incluso tratar de recuperarse del error automáticamente.

Los parámetros para esta función son el mismo objeto request con el que hemos venido tratando hasta aquí, y excepción, el cual es el objeto Exception real emitido por la función de vista.

process_exception() debe retornar ya sea None o un objeto HttpResponse.

- Si devuelve None, Django continuará procesando esta petición con el manejador de excepción incorporado en el framework.

- Si devuelve un objeto HttpResponse, Django usará esa respuesta en vez del manejador de excepción incorporado en el framework.

■ **Nota:** Django trae incorporado una serie de clases middleware (que se discuten en la sección siguiente) que hacen de buenos ejemplos. La lectura de su código debería darte una buena idea de la potencia del middleware.
También puedes encontrar una serie de ejemplos contribuidos por la comunidad en el wiki de Django: http://code.djangoproject.com/wiki/ContributedMiddleware

---

Middleware incorporado

Django viene con algunos middleware incorporados para lidiar con problemas comunes, los cuales discutiremos en las secciones que siguen.

Middleware de soporte para autenticación

Clase middleware: django.contrib.auth.middleware.AuthenticationMiddleware.

Este middleware permite el soporte para autenticación. Agrega el atributo request.user, que representa el usuario actual registrado, a todo objeto HttpRequest que se recibe.

Mira él *capítulo 12* para los detalles completos.

"Common" Middleware

Clase middleware: django.middleware.common.CommonMiddleware.

Este middleware agrega algunas conveniencias para los perfeccionistas:

- *Prohíbe el acceso a los agentes de usuario especificados en la configuración* DISALLOWED_USER_AGENTS: Si se especifica, esta configuración debería ser una lista de objetos de expresiones regulares compiladas que se comparan con el encabezado user-agent de cada petición que se recibe. Aquí está un pequeño ejemplo de un archivo de configuración:

```
import re

DISALLOWED_USER_AGENTS = (
    re.compile(r'^OmniExplorer_Bot'),
    re.compile(r'^Googlebot')
)
```

  Nota el import re, ya que DISALLOWED_USER_AGENTS requiere que sus valores sean expresiones regulares compiladas (es decir, el resultado de re.compile()). El archivo de configuración es un archivo común de Python, por lo tanto es perfectamente adecuado incluir sentencias import en él.

- *Realiza re-escritura de URL basado en las configuraciones* APPEND_SLASH y PREPEND_WWW: Si APPEND_SLASH es igual a True, las URLs que no poseen una barra al final serán redirigidas a la misma URL con una barra al final, a menos que el último componente en el path contenga un punto. De esta manera foo.com/bar es redirigido a foo.com/bar/, pero foo.com/bar/file.txt es pasado a través sin cambios.

  Si PREPEND_WWW es igual a True, las URLs que no poseen el prefijo "www." serán redirigidas a la misma URL con el prefijo "www.".

  Ambas opciones tienen por objeto normalizar URLs. La filosofía es que cada URL debería existir en un – y sólo un – lugar. Técnicamente la URL example.com/bar es distinta de example.com/bar/, la cual a su vez es distinta de www.example.com/bar/. Un motor de búsqueda indexador trataría de forma separada estas URLs, lo cual es perjudicial para la valoración de tu sitio en el motor de búsqueda, por lo tanto es una buena práctica normalizar las URLs.

- *Maneja ETags basado en la configuración* USE_ETAGS: ETags es una optimización a nivel HTTP para almacenar condicionalmente las páginas en la caché. Si USE_ETAGS es igual a True, Django calculará una ETag para cada petición mediante la generación de un hash MD5 del contenido de la página, y se hará cargo de enviar respuestas Not Modified, si es apropiado.

  Nota: también existe un middleware de GET condicional, que veremos en breve, el cual maneja ETags y hace algo más.

Middleware de compresión

Clase middleware: django.middleware.gzip.GZipMiddleware.

Este middleware comprime automáticamente el contenido para aquellos navegadores que comprenden la compresión gzip (todos los navegadores modernos). Esto puede reducir mucho la cantidad de ancho de banda que consume un servidor Web. La desventaja es que esto toma un poco de tiempo de procesamiento para comprimir las páginas.

Nosotros por lo general preferimos velocidad sobre ancho de banda, pero si tu prefieres lo contrario, solo habilita este middleware.

Middleware de GET condicional

Clase middleware: django.middleware.http.ConditionalGetMiddleware.

Este middleware provee soporte para operaciones GET condicionales. Si la respuesta contiene un encabezado Last-Modified o ETag, y la petición contiene If-None-Match o If-Modified-Since, la respuesta es reemplazada por una respuesta 304 ("Not modified"). El soporte para ETag depende de la configuración USE_ETAGS y espera que el encabezado ETag de la respuesta ya este previamente fijado. Como se señaló anteriormente, el encabezado ETag es fijado por el middleware Common.

También elimina el contenido de cualquier respuesta a una petición HEAD y fija los encabezados de respuesta Date y Content-Length para todas las peticiones.

Soporte para uso de proxy inverso (Middleware X-Forwarded-For)

Clase middleware: django.middleware.http.SetRemoteAddrFromForwardedFor.

Este es el ejemplo que examinamos en la sección anterior "Qué es middleware". Este establece el valor de request.META['REMOTE_ADDR'] basándose en el valor de request.META['HTTP_X_FORWARDED_FOR'], si este último está fijado. Esto es útil si estas parado detrás de un proxy inverso que provoca que cada petición REMOTE_ADDR sea fijada a 127.0.0.1.

☠ **Advertencia:** Este middleware *no* permite validar HTTP_X_FORWARDED_FOR.

Si no estás detrás de un proxy inverso que establece HTTP_X_FORWARDED_FOR automáticamente, no uses este middleware. Cualquiera puede inventar el valor de HTTP_X_FORWARDED_FOR, y ya que este establece REMOTE_ADDR basándose en HTTP_X_FORWARDED_FOR, significa que cualquiera puede falsear su dirección IP.

Solo usa este middleware cuando confíes absolutamente en el valor de HTTP_X_FORWARDED_FOR.

Middleware de soporte para sesiones

Clase middleware: django.contrib.sessions.middleware.SessionMiddleware.

Este middleware habilita el soporte para sesiones. Mira el *capítulo 14* para más detalles.

Middleware de cache de todo el sitio

Clase middleware: django.middleware.cache.CacheMiddleware.

Este middleware almacena en la cache cada página impulsada por Django. Este se analizó en detalle en el *capítulo 15*.

Middleware de transacción

Clase middleware: django.middleware.transaction.TransactionMiddleware.

Este middleware asocia un COMMIT o ROLLBACK de la base de datos con una fase de petición/respuesta. Si una vista de función se ejecuta con éxito, se emite un COMMIT. Si la vista provoca una excepción, se emite un ROLLBACK.

El orden de este middleware en la pila es importante. Los módulos middleware que se ejecutan fuera de este, se ejecutan con commit-on-save – el comportamiento por omisión de Django. Los módulos middleware que se ejecutan dentro de este (próximos al final de la pila) estarán bajo el mismo control de transacción que las vistas de función.

Mira el Apéndice B para obtener más información sobre las transacciones de base de datos.

Middleware "X-View"

Clase middleware: django.middleware.doc.XViewMiddleware.

Este middleware envía cabeceras HTTP X-View personalizadas a peticiones HEAD que provienen de direcciones IP definidas en la configuración INTERNAL_IPS. Esto es usado por el sistema automático de documentación de Django.

¿Qué sigue?

Los desarrolladores Web y los diseñadores de esquemas de bases de datos no siempre tienen el lujo de comenzar desde cero. En el *próximo capítulo*, vamos a cubrir el modo de integrar Django con sistemas existentes, tales como esquemas de bases de datos que has heredado de la década de los 80.

CAPÍTULO 18

Integración con base de datos y aplicaciones existentes

Django es el más adecuado para el desarrollo denominado de campo verde – es decir, comenzar proyectos desde cero, como si estuviéramos construyendo un edificio en un campo fresco de pasto verde. Pero a pesar de que Django favorece a los proyectos iniciados desde cero, es posible integrar el framework con bases de datos y aplicaciones existentes. Este capítulo explica algunas de las estrategias de integración.

Integración con una base de datos existente

La capa de base de datos de Django genera esquemas SQL desde código Python – pero con una base de datos existente, tú ya tienes los esquemas SQL. En tal caso, necesitas crear modelos para tus tablas de la base de datos existente. Para este propósito, Django incluye una herramienta que puede generar el código del modelo leyendo el diseño de las tablas de la base de datos. Esta herramienta se llama inspectdb, y puedes llamarla ejecutando el comando manage.py inspectdb.

Usando inspectdb

La utilidad inspectdb realiza una introspección de la base de datos a la que apunta tu archivo de configuración, determina una representación del modelo que usará Django para cada una de tus tablas, e imprime el código Python del modelo a la salida estándar.

Esta es una guía de un proceso típico de integración con una base de datos existente desde cero. Las únicas suposiciones son que Django está instalado y tienes una base de datos existente.

1. Crea un proyecto Django ejecutando django-admin.py startproject mi_sitio (donde mi_sitio es el nombre de tu proyecto). Usaremos mi_sitio como el nombre de nuestro proyecto, en este ejemplo.

2. Edita el archivo de configuración en ese proyecto, mi_sitio/settings.py, para decirle a Django cuáles son los parámetros de conexión a tu base de datos y cuál es su nombre. Específicamente, provee las configuraciones de DATABASE_NAME, DATABASE_ENGINE, DATABASE_USER, DATABASE_PASSWORD, DATABASE_HOST, y DATABASE_PORT. (Ten en cuenta que algunas de estas configuraciones son opcionales, ya que dependen de la base de datos a usar. Mira él *capítulo 5* para más información).

3. Crea una aplicación dentro de tu proyecto ejecutando python mi_sitio/manage.py startapp myapp (donde myapp es el nombre de tu aplicación). Usaremos myapp como el nombre de aplicación aquí.

4. Ejecuta el comando python mi_sitio/manage.py inspectdb. Esto examinará las tablas en la base de datos DATABASE_NAME e imprimirá para cada tabla el modelo de clase generado. Dale una mirada a la salida para tener una idea de lo que puede hacer inspectdb.

5. Guarda la salida en el archivo models.py dentro de tu aplicación usando la redirección de salida estándar de la shell:

```
python mi_sitio/manage.py inspectdb > mi_sitio/myapp/models.py
```

6. Edita el archivo mi_sitio/myapp/models.py para limpiar los modelos generados y realiza cualquier personalización necesaria. Te daremos algunas sugerencias para esto en la siguiente sección.

Limpiar los modelos generados

Como podrías esperar, la introspección de la base de datos no es perfecta, y necesitarás hacer una pequeña limpieza al código del modelo resultante. Aquí hay algunos apuntes para lidiar con los modelos generados:

- Cada tabla de la base de datos es convertida en una clase del modelo (es decir, hay un mapeo de uno-a-uno entre las tablas de la base de datos y las clases del modelo). Esto significa que tendrás que refactorizar los modelos para tablas con relaciones muchos-a-muchos en objetos ManyToManyField.

- Cada modelo generado tiene un atributo para cada campo, incluyendo campos de clave primaria id. Sin embargo, recuerda que Django agrega automáticamente un campo de clave primaria id si un modelo no tiene una clave primaria. Por lo tanto, es necesario remover cualquier línea que se parezca a ésta:

  ```
  id = models.IntegerField(primary_key=True)
  ```

  No solo estas líneas son redundantes, sino que pueden causar problemas si tu aplicación agregara *nuevos* registros a estas tablas. El comando inspectdb no puede detectar si un campo es autoincrementado, así que está en tí cambiar esto a AutoField, si es necesario.

- Cada tipo de campo (ej., CharField, DateField) es determinado mirando el tipo de la columna de la base de datos (ej., VARCHAR, DATE). Si inspectdb no puede mapear un tipo de columna a un tipo de campo del modelo, usará TextField e insertará el comentario Python 'This field type is a guess.' a continuación del campo en el modelo generado. Mantén un ojo en eso, y cambia el tipo de campo adecuadamente si es necesario.

  Si un campo en tu base de datos no tiene un buen equivalente en Django, con seguridad puedes dejarlo fuera. La capa de modelo de Django no requiere que incluyas todos los campos de tu(s) tabla(s).

- Si un nombre de columna de tu base de datos es una palabra reservada de Python (como pass, class o for), inspectdb agregará '_field' al nombre del

atributo y establecerá el atributo db_column al nombre real del campo (ej., pass, class, o for).

Por ejemplo, si una tabla tiene una columna INT llamada for, el modelo generado tendrá un campo como este:

```
for_field = models.IntegerField(db_column='for')
```

inspectdb insertará el comentario Python 'Field renamed because it was a Python reserved word.' a continuación del campo.

- Si tu base de datos contiene tablas que hacen referencia a otras tablas (como la mayoría de las bases de datos lo hacen), tal vez tengas que re-acomodar el orden de los modelos generados, de manera que los modelos que hacen referencia a otros modelos estén ordenados apropiadamente. Por ejemplo, si un modelo Libro tiene una ForeignKey al modelo Autor, el modelo Autor debe ser definido antes del modelo Libro. Si necesitas crear una relación en un modelo que todavía no está definido, puedes usar el nombre del modelo, en vez del objeto modelo en sí.

- *inspectdb* detecta claves primarias para PostgreSQL, MySQL y SQLite. Es decir, inserta primary_key=True donde sea necesario. Para otras bases de datos, necesitarás insertar primary_key=True para al menos un campo en cada modelo, ya que los modelos Django requieren tener un campo primary_key=True.

- La detección de claves foráneas sólo funciona con PostgreSQL y con ciertos tipos de tablas MySQL. En otros casos, los campos de clave foránea serán generados como campos IntegerField, asumiendo que la columna de clave foránea fue una columna INT.

Integración con un sistema de autentificación

Es posible integrar Django con un sistema de autentificación existente – otra fuente de nombres de usuario y contraseñas o métodos de autentificación.

Por ejemplo, tu compañía ya puede tener una configuración LDAP que almacena un nombre de usuario y contraseña para cada empleado. Sería una molestia tanto para el administrador de red como para los usuarios, si cada uno de ellos tiene cuentas separadas en LDAP y en las aplicaciones basadas en Django.

Para manejar situaciones como ésta, el sistema de autentificación de Django te permite conectarte con otras fuentes de autentificación. Puedes anular el esquema por omisión de Django basado en base de datos, o puedes usar el sistema por omisión en conjunto con otros sistemas.

Especificar los back-ends de autentificación

Detrás de escena, Django mantiene una lista de "back-ends de autentificación" que utiliza para autentificar. Cuando alguien llama a django.contrib.auth.authenticate() (como se describió en el *capítulo 12*), Django intenta autentificar usando todos sus back-ends de autentificación. Si el primer método de autentificación falla, Django intenta con el segundo, y así sucesivamente, hasta que todos los back-ends han sido intentados.

La lista de back-ends de autentificación a usar se especifica en la configuración AUTHENTICATION_BACKENDS. Ésta debe ser una tupla de nombres de ruta Python que apuntan a clases que saben cómo autentificar. Estas clases pueden estar en cualquier lugar de tu ruta Python.

Por omisión, AUTHENTICATION_BACKENDS contiene lo siguiente:

```
('django.contrib.auth.backends.ModelBackend',)
```

Ese es el esquema básico de autentificación que verifica la base de datos de usuarios de Django.

El orden de AUTHENTICATION_BACKENDS se tiene en cuenta, por lo que si el mismo usuario y contraseña son válidos en múltiples back-ends, Django detendrá el procesamiento en la primera coincidencia positiva.

Escribir un back-end de autentificación

Un back-end de autentificación es un clase que implementa dos métodos: get_user(id) y authenticate(**credentials).

El método get_user recibe un id – el cual podría ser un nombre de usuario, un ID de la base de datos o cualquier cosa – y devuelve un objeto User.

El método authenticate recibe credenciales como argumentos de palabras clave. La mayoría de las veces se parece a esto:

```
class MyBackend(object):
    def authenticate(self, username=None, password=None):
        # Verifica el nombre de usuario/contraseña y devuelve el usuario.
```

Pero podría también autentificar un *token*, como se muestra a continuación:

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Verifica el token y devuelve el usuario.
```

De cualquier manera, authenticate debe verificar las credenciales que recibe, y debe retornar un objeto User que coincide con esas credenciales, si las credenciales son válidas. Si no son válidas, debe retornar None.

El sistema de administración de Django está altamente acoplado a su propio objeto User respaldado por base de datos descripto en el *capítulo 12*. La mejor manera de lidiar con esto es crear un objeto User de Django para cada usuario que existe en tu back-end (ej., en tu directorio LDAP, tu base de datos SQL externa, etc.). De cualquier manera puedes escribir un script para hacer esto por adelantado o tu método de autentificación puede hacerlo la primera vez que el usuario ingresa al sistema.

Aquí está un ejemplo de back-end que autentifica contra unas variables de usuario y contraseña definidas en tu archivo settings.py y crea un objeto User de Django la primera vez que un usuario se autentifica:

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend(object):
    """
    Autentificación contra la configuración ADMIN_LOGIN y ADMIN_PASSWORD.
    Usa el nombre de login, y el hash del password. Por ejemplo:
    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD =
        'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
    """
    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Crea un nuevo usuario. Nota que podemos fijar un password
                # para cualquiera, porque este no será comprobado; el password
                # de settings.py lo hará.
                user = User(username=username, password='get from settings.py')
                user.is_staff = True
                user.is_superuser = True
                user.save()

            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Integración con aplicaciones web existentes

Es posible ejecutar una aplicación Django en el mismo servidor de una aplicación impulsada por otra tecnología. La manera más directa de hacer esto es usar el archivo de configuración de Apache, httpd.conf, para delegar patrones de URL diferentes a distintas tecnologías (Nota que él *capítulo 12* cubre el despliegue con Django en Apache/wsgi_python, por lo tanto tal vez valga la pena leer ese capítulo antes de intentar esta integración).

La clave está en que Django será activado para un patrón particular de URL sólo si tu archivo httpd.conf lo dice. El despliegue por omisión explicado en el *capítulo 12*, asume que quieres que Django impulse todas las páginas en un dominio particular:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mi_sitio.settings
    PythonDebug On
</Location>
```

Aquí, la línea <Location "/"> significa "maneja cada URL, comenzando en la raíz", con Django.

Esta perfectamente bien limitar esta directiva <Location> a cierto árbol de directorio. Por ejemplo, digamos que tienes una aplicación PHP existente que impulsa la mayoría de las páginas en un dominio y quieres instalar el sitio de administración de Django en /admin/ sin afectar el código PHP. Para hacer esto, sólo configura la directiva <Location> a /admin/:

```
<Location "/admin/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mi_sitio.settings
    PythonDebug On
</Location>
```

Con esto en su lugar, sólo las URLs que comiencen con /admin/ activarán Django. Cualquier otra página usará cualquier infraestructura que ya exista.

Nota que adjuntar Django a una URL calificada (como /admin/ en el ejemplo de esta sección) no afecta a Django en el análisis de las URLs. Django trabaja con la URL absoluta (ej., /admin/people/person/add/), no con una versión "recortada" de la URL (ej., /people/person/add/). Esto significa que tu URLconf raíz debe incluir el prefijo /admin/.

¿Qué sigue?

Si tu idioma nativo es el inglés –cosa que gracias a los traductores ya no es necesaria para leer este libro– quizás no te hayas enterado de una de las más fantásticas características de la interfaz de administración: ¡está disponible en más de 50 idiomas distintos! Esto es posible gracias al framework de internacionalización de Django (y el duro trabajo de los traductores voluntarios de Django). Él *Capítulo19* explica cómo usar este framework para crear sitios Django localizados.

CAPÍTULO 19

Internacionalización

Django fue desarrollado originalmente en medio de los Estados Unidos (literalmente; Lawrence, Kansas, se halla a menos de 40 millas del centro geográfico de la porción continental, de los Estados Unidos). Sin embargo, como la mayoría de los proyectos de código abierto (*open source)*, la comunidad de Django creció hasta incluir a gente de todo el mundo. A medida que la comunidad fue tornándose más diversa, la *internacionalización* y la *localización* fueron tomando una importancia creciente. Debido a que muchos desarrolladores tienen, en el mejor de los casos, una comprensión difusa de dichos términos vamos a definirlos brevemente.

El término **Internacionalizació**n se refiere al proceso de diseñar programas para el uso potencial de cualquier localidad. Esto incluye marcas de texto (tales como elementos de la interfaz de usuario o mensajes de error) para su futura traducción, abstrayendo la visualización de fechas y horarios de manera que sea posible respetar diferentes estándares locales, proveer soporte para diferentes zonas horarias y en general el asegurarse de que el código no contenga ninguna suposición acerca de la ubicación de sus usuarios. Encontrarás a menudo "internacionalización" abreviada como I18N (el número 18 se refiere al número de letras omitidas entre la "I" inicial y la "N" final).

El término **Localización** se refiere al proceso específico de traducir un programa internacionalizado para su uso en una localidad en particular. Encontrarás a menudo "localización" abreviada como L10N.

Django en si está totalmente internacionalizado; todas las cadenas están marcadas para su traducción y existen variables de configuración que controlan la visualización de valores locales dependientes como fechas y horarios. Django también incluye más de 40 archivos de localización. Si no hablas inglés en forma nativa, existe una buena probabilidad de que Django ya se encuentre traducido a tu idioma nativo.

El mismo framework de internacionalización usado para esas localizaciones está disponible para que lo uses en tu propio código y plantillas.

En resumen, necesitarás agregar una cantidad mínima de ganchos a tu código Python y a tus plantillas. Estos ganchos reciben el nombre de *cadenas de traducción*. Los mismos le indican a
Django "Este texto debe ser traducido al idioma del usuario final si existe una traducción a dicho idioma de ese texto."

Django se encarga de usar estos ganchos para traducir las aplicaciones Web "al vuelo" de acuerdo a las preferencias de idioma del usuario.
Esencialmente, Django hace dos cosas:

- Le permite a los desarrolladores y autores de plantillas especificar qué partes de sus aplicaciones deben ser traducibles.

- Usa esta información para traducir las aplicaciones Web para usuarios particulares de acuerdo a sus preferencias de idioma.

Nota: La maquinaria de traducción de Django usa gettext de GNU (http://www.gnu.org/software/gettext/) vía el módulo estándar gettext incluido en Python.

SI NO NECESITAS USAR INTERNACIONALIZACIÓN

Los ganchos de internacionalización de Django se encuentran activos por omisión, lo cual incurre en una pequeña sobrecarga. Si no utilizas la internacionalización, deberías establecer ***USE_I18N = False*** en tu archivo de configuración. Si USE_I18N tiene el valor False Django implementará algunas optimizaciones de manera de no cargar la maquinaria de localización.
Probablemente querrás también eliminar django.core.context_processors.i18n de tu variable de configuración TEMPLATE_CONTEXT_PROCESSORS.

Los tres pasos para usar la internacionalización en aplicaciones Django son:

1. Especifica las cadenas de traducción en el código Python y en las plantillas.
2. Implementa las traducciónes para esas cadenas, en cualquiera de los lenguajes que quieras soportar.
3. Activa el middleware local en la configuración de Django.

Cubriremos cada uno de estos pasos detalladamente.

Especifica las cadenas de traducción en código Python

Las cadenas de traducción especifican “Este texto debería ser traducido.” dichas cadenas pueden aparecer en tu código Python y en tus plantillas. Es tú responsabilidad marcar las cadenas traducibles; el sistema sólo puede traducir cadenas sobre las que está al tanto.

Funciones estándar de traducción

Las cadenas de traducción se especifican usando la función ugettext(). Este por convención usa el alias _ (guion bajo), como un atajo para importar la función.

En este ejemplo, el texto "Bienvenido a mi sitio." está marcado como una cadena de traducción, usando el alias _:

```
from django.utils.translation import ugettext as _
from django.http import HttpResponse

def mi_vista(request):
    salida = _("Bienvenido a mi sitio.")
    return HttpResponse(salida)
```

La función django.utils.translation.gettext() es idéntica a _(). Este ejemplo es idéntico al anterior:

```
from django.utils.translation import ugettext

def mi_vista(request):
    salida = ugettext("Bienvenido a mi sitio.")
    return HttpResponse(salida)
```

La mayoría de los desarrolladores prefiere usar el alias _(), debido a que es más corto.

La librería estándar de Python gettext instala un modulo _() en el espacio de nombres global, como un alias para gettext(). En Django hemos decidido no seguir esta práctica por un par de razones:

Para el apoyo internacional del conjunto de caracteres (Unicode), La función django.utils.translation.ugettext es más útil que gettext(). Algunas veces puedes utilizar por defecto la función django.utils.translation.ugettext_lazy como método para traducir un archivo en particular. Sin usar el _() en el espacio de nombres global, el desarrollador tiene que pensar acerca de cuál es la función más apropiada para usar en la traducción.

El carácter de guion bajo (_) es usado para representar "el resultado previo" en el interprete interactivo Python y en las pruebas de documentación. Instalar una función global _() causa interferencia. Explícitamente importando ugettext() como _() evitamos este problema.

La traducción funciona también sobre valores computados. Este ejemplo es idéntico a los dos anteriores:

```
def mi_vista(request):
    palabras = ['Bienvenido', 'a', 'mi', 'sitio.']
    salida = _(' '.join(palabras))
    return HttpResponse(salida)
```

La traducción funciona también sobre variables. De nuevo, este es otro ejemplo idéntico:

```
def mi_vista(request):
    sentencia = 'Bienvenido a mi sitio.'
    salida = _(sentencia)
    return HttpResponse(salida)
```

(Algo a tener en cuenta cuando se usan variables o valores computados, como se veía en los dos ejemplos previos, es que la utilidad de detección de cadenas de traducción de Django, makemessages, no será capaz de encontrar esas cadenas. Trataremos el comando django-admin.py makemessages más adelante).

Las cadenas que le pasas a _() o ugettext() pueden contener marcadores de posición especificados con la sintaxis estándar de interpolación de cadenas de Python con nombres, por ejemplo:

```
def mi_vista(request, m, d):
    salida = _('Hoy es %(mes)s %(dia)s.') % {'mes': m, 'dia': d}
    return HttpResponse(salida)
```

Esta técnica permite que las traducciónes específicas de cada idioma reordenen el texto de los marcadores de posición. Por ejemplo, una traducción al inglés podría ser "Today is November 26.", mientras que una traducción al español podría ser "Hoy es 26 de Noviembre.", usando marcadores de posición para el mes y el dia para intercambiarlos.

Por esta razón, deberías usar interpolación de cadenas con nombres (por ejemplo %(dia)s) en lugar de interpolación posicional (por ejemplo %s o %d). Si usas interpolación posicional las traducciónes no serán capaces de reordenar el texto de los marcadores de posición.

Comentarios para traducciónes

Si te gustaría darle indicios a los traductores acerca de una cadena traducible, puedes añadir un comentario, usando un prefijo con la palabra clave Translators en la línea anterior a la cadena, por ejemplo:

```
def mi_vista(request):
    # Translators: Este mensaje aparece en la página de inicio únicamente.
    output = ugettext("Bienvenidos a mi sitio.")
```

El comentario después aparecerá en el fichero resultante .po asociado, con la traducción compilada y localizado debajo de él mensaje traducido, como en el siguiente fragmento:

```
#. Translators: Este mensaje aparece en la página de inicio
únicamente.
# path/to/python/file.py:123
msgid "Bienvenidos a mi sitio."
msgstr ""
```

Esto también funciona en plantillas. Así como en el código Python, las notas para traductores pueden especificarse usando comentarios en las plantillas de la siguiente forma.

```
{% comment %}  Translators: Verbo de la vista {% endcomment %}

{% trans "Vista" %}

{% comment %}Translators: Introducción corta sobre publicidad{% endcomment %}
    <p>{% blocktrans %}Traducir varias líneas de forma literal.{% endblocktrans %}</p>
```

o con etiquetas {# ... #} uno por línea, para construir comentarios:

```
{# Translators: Etiqueta de un botón de búsqueda #}
<button type="submit">{% trans "Ir" %}</button>

{# Translators: Éste es texto de la plantilla base #}
{% blocktrans %}Bloque ambiguo de texto traducible{% endblocktrans %}
```

Como complemento, este es un fragmento de el resultado en un archivo .po.

```
#. Translators: Verbo de la vista
# ruta/a/plantilla/archivo.html:10
msgid "Vista"
msgstr ""
```

```
#. Translators: Introducción corta sobre publicidad
# path/to/template/file.html:13
msgid ""
"Traducir varias líneas"
"literal."
msgstr ""

# ...

#. Translators: Etiqueta de un botón de búsqueda
# ruta/a/plantilla/archivo.html:100
msgid "Ir"
msgstr ""

#. Translators: Éste es texto de la plantilla base
# ruta/a/plantilla/archivo.html:103
msgid "bloque ambiguo de texto traducible"
msgstr ""
```

Marcando cadenas como no-op

Usa la función django.utils.translation.ugettext_noop() para marcar una cadena como una cadena de traducción sin realmente traducirla en ese momento. Las cadenas así marcadas no son traducidas sino hasta el último momento que sea posible.

Usa este enfoque si deseas tener cadenas constantes que deben ser almacenadas en el idioma original – tales como cadenas en una base de datos, pero que deben ser traducidas en el último momento posible, por ejemplo cuando la cadena es presentada al usuario.

Traducción perezosa

Usa la función django.utils.translation.ugettext_lazy() para traducir cadenas en forma perezosa – cuando el valor es accedido en lugar de cuando se llama a la función ugettext_lazy().

Por ejemplo, para marcar el atributo texto_ayuda de un campo como traducible, haz lo siguiente:

```
from django.db import models
from django.utils.translation import ugettext_lazy

class Mimodelo(models.Model):
    nombre = models.CharField(help_text=ugettext_lazy('Este es el texto de ayuda'))
```

En este ejemplo, ugettext_lazy() almacena una referencia perezosa a la cadena – no al verdadero texto traducido. La traducción en si misma se llevará a cabo cuando sea usada en un contexto de cadena, tal como el renderizado de una plantilla en el sitio de administración de Django.

El resultado de llamar a ugettext_lazy() puede ser usado donde se necesite usar una cadena en Unicode (un objeto del tipo Unicode) en Python. Si tratas de usar un bytestring donde se espera un objeto str, las cosas no funcionaran como esperabas,

ya que la función ugettext_lazy() no puede convertir en si mismo objetos bytestring. Sin embargo puedes usar cadenas Unicode dentro de de cualquier bytestring esto es consistente con el comportamiento normal de Python.

Por ejemplo:

```
# Esto está bien: poner un marcador Unicode dentro de una cadena Unicode.
“Hola %s" % ugettext_lazy("gente")

# Esto no funcionara, ya que no se puede insertar un objeto Unicode
# dentro de un bytestring.
b"Hola %s" % ugettext_lazy("gente")
```

Si ves en la salida algo como esto "hola <django.utils.functional...>", estas tratando de insertar el resultado de ugettext_lazy() en un bytestring. Se trata de un error en tu código.

Si no te gusta el nombre largo gettext_lazy puedes simplemente crear un alias _ (guión bajo) para el mismo, de la siguiente forma:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Mimodelo(models.Model):
   titulo = models.CharField(help_text=_('Texto de ayuda'))
```

Usa siempre traducciónes perezosas en modelos Django (de lo contrario no serán traducidos correctamente para cada usuario). Y es una buena idea agregar también traducciónes de los nombres de campos y nombres de tablas. Esto significa que también puedes especificar las opciones verbose_name y verbose_name_plural como traducibles en forma explícita en la clase Meta, así:

```
from django.utils.translation import ugettext_lazy as _
from biblioteca.models import Libro

class Libro(models.Model):
   titulo = models.CharField(_('titulo'), help_text=_('Escribe el título del libro'))
    #..

   class Meta:
      verbose_name = _('libro')
      verbose_name_plural = _('libros')
```

Pluralización

Usa la función django.utils.translation.ungettext() para especificar mensajes que tienen formas singular y plural distintas.

ungettext toma tres argumentos: la cadena de traducción_ en singular, la cadena de traducción en plural y el numero de objetos.
Esta función es útil cuando se necesita que una aplicación Django sea localizable por los lenguajes donde el número y la complejidad de las formas que toma el plural es mayor que las dos formas usadas en el inglés (objeto para el singular y objetos para el plural, para casos donde count es diferente a uno, sin distinción de su valor.)

Por ejemplo:

```
from django.utils.translation import ungettext
from django.http import HttpResponse

def ejemplo_pluralizacion(request, count):
    page = ungettext(
          'Este es %(count)d objeto',
         'Estos son %(count)d objetos',
    count) % {
        'count': count,
    }

    return HttpResponse(page)
```

En este ejemplo el número de objetos es pasado al lenguaje de traducción como la variable count.

Observa que la pluralización es complicada y funciona diferente en cada lenguaje Comparar count con 1 no es siempre la regla correcta. Este código parece sofisticado, pero producirá resultados incorrectos en algunos lenguajes:

No intentes implementar tu propia lógica singular o plural, no sería correcto. En un caso como este, considera hacer algo así como lo siguiente:

Nota: Al usar ungettext (), asegúrate de utilizar un único nombre para cada variable extrapolada incluida literalmente. En los ejemplos anteriores, nota cómo utilizamos la variable de Python nombre en ambas cadenas de la traducción. En el siguiente ejemplo, nota que además de ser incorrecto en algunos lenguajes según lo observado anteriormente, fallara:

```
text = ungettext(
    'Este es %(count)d %(nombre)s disponible.',
    'Estos son %(count)d %(nombre_plural)s disponibles.',
    count
) % {
    'count': Libro.objects.count(),
    'nombre': Report._meta.verbose_name,
    'nombre_plural': Libro._meta.verbose_name_plural
}
```

Al ejecutar el comando django-admin compilemessages este retornara un error:

```
a format specification for argument 'nombre', as in 'msgstr[0]',
doesn't exist in 'msgid'
```

Cadenas de traducción en plantillas

Las traducciónes en las plantillas de Django usan dos etiquetas de plantilla y una sintaxis ligeramente diferente a la del código Python. Para que tus plantillas puedan acceder a esas etiquetas coloca {% load i18n %} al principio de tu plantilla.

La etiqueta de plantilla {% trans %} marcan una cadena para su traducción, (encerradas por comillas simples o dobles) o el contenido de una variable:

```
<title>{% trans "Este es el titulo." %}</title>
<title>{% trans soyunavariable %}</title>
```

Si la opción *noop* está presente, las operaciones de búsqueda de variables todavía ocurre, pero se salta la traducción. Esto es útil cuando el contenido "de fuera" requiere la traducción en el futuro.

```
<title>{% trans "valor" noop %}</title>
```

No es posible mezclar en las plantillas variables dentro de cadenas sin la etiqueta {% trans %} . Si tu traducción requiere variables (marcadores de posición) puedes usar por ejemplo {% blocktrans %}.

```
{% blocktrans %}
   Esta cadena tiene un {{ valor }} dentro.
{% endblocktrans %}
```

Para traducir una expresión de plantilla – por ejemplo, usando filtros de plantillas – necesitas asociar la expresión a una variable local que será la que se usará dentro del bloque de traducción:

```
{% blocktrans with valor|filter as variable %}
   Esta tiene una {{ variable }} dentro.
{% endblocktrans %}
```

Si necesitas asociar más de una expresión dentro de una etiqueta blocktrans, separa las partes con and:

```
{% blocktrans with libro|titulo as mi_libro and autor|titulo as mi_autor %}
   Este es {{ mi_libro }} por {{ mi_autor}}
{% endblocktrans %}
```

Para pluralizar, especifica tanto la forma singular como la plural con la etiqueta {% plural %} la cual aparece dentro de {% blocktrans %} y {% endblocktrans %}, por ejemplo:

```
{% blocktrans count list|length as counter %}
      Hay únicamente {{ nombre }} objeto.

{% plural %}
   Hay  {{ counter }} {{ nombre }} objetos.
{% endblocktrans %}
```

Internamente, todas las traducciónes en bloque y en línea usan las llamadas apropiadas a ugettext/ ungettext.

Cuando usas RequestContext , tus plantillas tienen acceso a tres variables específicas relacionadas con la traducción:

- {{ LANGUAGES }} es una lista de tuplas en las cuales el primer elemento es el código de idioma y el segundo es el nombre y escrito usando el mismo).

- {{ LANGUAGE_CODE }} es el idioma preferido del usuario actual, expresado como una cadena (por ejemplo en-us). (Consulta la sección "3. Cómo descubre Django la preferencia de idioma" para información adicional).

- {{ LANGUAGE_BIDI }} es el sistema de escritura del idioma actual. Si el valor es True, se trata de un idioma derecha-a-izquierda (por ejemplo hebreo, árabe). Si el valor es False, se trata de de un idioma izquierda-a-derecha (por ejemplo inglés, francés, alemán).

  Si no usas la extensión RequestContext, puedes usar estos valores, con estas tres etiquetas de plantilla:

```
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

Estas etiquetas también requieren de {% load i18n %}.

Los ganchos pasa las traducciónes están disponibles en el interior de cualquier etiqueta de bloque de plantilla que acepte cadenas constantes. En dichos casos basta con que uses la sintaxis _() para especificar que es una cadena para traducción, por ejemplo:

```
{% alguna_etiqueta_especial _("Pagina no encontrada") value|yesno:_("si,no") %}
```

En este caso tanto la etiqueta como el filtro verán la cadena ya traducida (en otras palabras la cadena es traducida *antes* de ser pasada a las funciones de manejo de etiquetas), de manera que no necesitan estar preparadas para manejar traducción.

Nota: En este ejemplo, la infraestructura de traducciónes pasa la cadena "si,no" como una sola y no como cadenas individuales "si" y "no". La cadena de traducción necesitará contener la coma de modo que el filtro de análisis sepa dividir los argumentos. Por ejemplo, una traductor Alemán podría traducir la cadena "si,no" como "ja,nein" (manteniendo la coma intacta).

Trabajando con objetos en traducción perezosas

El uso de ugettext_lazy() y ungettext_lazy() para marcar cadenas en modelos y funciones de utilidad general es una operación muy común, Cuando trabajamos con estos objetos en cualquier parte de nuestro código, debemos asegurarnos de no convertir nuestro código accidentalmente en cadenas, para ello necesitamos asegurarnos de convertir las cadenas lo más tarde posible (de modo que la traducción correcta surta efecto). Esto hace necesario el uso de unas par de funciones de ayuda.

- string_concat()
- allow_lazy()

Juntando cadenas con: string_concat()

El método estándar de Python join, usado para juntar cadenas (''.join([...])) no trabaja en listas que contienen traducciónes perezosas. En su lugar debes usar django.utils.translation.string_concat(), el cual crea un objeto perezoso que concatena el contenido *y* convierte las cadenas únicamente cuando el resultado es incluido en la cadena.

Por ejemplo:

```
from django.utils.translation import string_concat

 # ...
nombre = ugettext_lazy(u'John Lennon')
instrumento = ugettext_lazy(u'guitarra')
resultado = string_concat([nombre, ': ', instrumento])
```

En este caso, la traducción perezosa en la variable resultado únicamente convierte la cadena cuando resultado es usado en una cadena (usualmente cuando se renderiza la plantilla)

El decorador allow_lazy()

Django ofrece muchas funciones útiles (agrupadas en el paquete django.utils) que toman una cadena como su primer argumento y hacen algo con la cadena. Estas funciones son usadas por los filtros de plantillas así como directamente en el código.

Si escribes tus propias funciones y te ocupas de las traducciónes, lo más seguro es que te has encontrado con este problema, ¿Qué hacer cuando el primer argumento es un objeto perezoso de una traducción? Si necesitas convertirlo a una cadena inmediatamente, porque tal vez necesites usarlo fuera de una función de vista (y por lo tanto la configuración actual no funciona)

Para casos como este, usa el decorador django.utils.functional.allow_lazy() El cual modifica la función *si* es llamado con una traducción perezosa como su primer argumento, de esta forma el decorador demora la función hasta que necesite convertirse a cadena.

Por ejemplo:

```
from django.utils.functional import allow_lazy

def fancy_utility_function(s, ...):
    # Hace la conversión en cadena 's'
    # ...
    fancy_utility_function = allow_lazy(fancy_utility_function, Unicode)
```

El decorador allow_lazy() toma, en adición a la función que decora, un numero extra de argumentos (*args) especificando el tipo que la función original debe devolver. Usualmente, es suficiente incluir Unicode y asegurarse que la función devuelva únicamente cadenas Unicode.

Usando este decorador significa que puedes escribir tu función y asumir que la entrada es justamente una cadena, después solo agrega el soporte para traducciónes de objetos perezosos al final.

Como crear archivos de idioma

Una vez que hayas etiquetado tus cadenas para su posterior traducción, necesitas escribir (u obtener) las traducciónes propiamente dichas. En esta sección explicaremos como es que eso funciona.

RESTRICCIONES LOCALES

Django no soporta *Localización* en una aplicación local para la cual Django en sí mismo no ha sido traducido. En este caso solo ignorara el archivo de traducción. Si intentas hacer esto y Django lo soporta, inevitablemente veras una mezcla de cadenas traducidas (de tu aplicación) y cadenas en Ingles (de Django mismo) Si quieres soportar el idioma de tu aplicación en un formato local, que no sea parte de Django, necesitarás hacer por lo menos una traducción mínima del núcleo de Django.

Creando los archivos de mensajes

El primer paso es crear un *archivo de mensajes* para un nuevo idioma. Un archivo de mensajes es un archivo de texto común que representa un único idioma que contiene todas las cadenas de traducción disponibles y cómo deben ser representadas las mismas en el idioma en cuestión. Los archivos de mensajes tienen una extensión .po.

Django incluye una herramienta, bin/make-messages, que automatiza la creación y el mantenimiento de dichos archivos.

Para crear o actualizar un archivo de mensajes, ejecuta este comando:

```
bin/make-messages.py -l de
```

Donde *de* es el código de idioma para el archivo de mensajes que deseas crear. El código de idioma en este caso está en formato locale. Por ejemplo, el mismo es pt_BR para portugués de Brasil y de_AT para alemán de Austria. Echa un vistazo a los códigos de idioma en el directorio django/conf/locale/ para ver cuáles son los idiomas actualmente incluidos.

El script debe ser ejecutado desde una de tres ubicaciones:

- El directorio raíz de tu proyecto Django.
- El directorio raíz de tu aplicación Django.
- El directorio raíz django (no una copia de git, sino el que se halla referenciado por $PYTHONPATH o que se encuentra en algún punto debajo de esa ruta. Este es únicamente relevante si estas creando una traducción para Django mismo.

El script recorre completamente el árbol en el cual es ejecutado y extrae todas las cadenas marcadas para traducción. Crea (o actualiza) un archivo de mensajes en el directorio locale/LANG/LC_MESSAGES. En el ejemplo de, el archivo será locale/de/LC_MESSAGES/django.po.

Por defecto django-admin.py makemessages examina cada archivo que tenga una extensión .html. En el caso de que quieras sobrescribirá el valor por default usa la opción --extension o -e para especificar la extensión del archivo a examinar:

```
django-admin.py makemessages -l de -e txt
```

Separa múltiples extensiones con comas y/o usa -e o --extensión varias veces:

```
django-admin.py makemessages -l de -e html,txt -e xml
```

Al crear traducciónes para catálogos JavaScript (el cual cubriremos más adelante en este capítulo), necesitaras usar un dominio especial 'djangojs' **no** -e js.

¿Sin gettext?

Si no tienes instaladas las utilidades gettext, django-admin.py makemessages creará archivos vacíos. Si te encuentras ante esa situación debes o instalar dichas utilidades o simplemente copiar el archivo de mensajes de inglés (conf/locale/en/LC_MESSAGES/django.po) y usar el mismo como un punto de partida; se trata simplemente de un archivo de traducción vacío, que te servirá para crear el tuyo.

¿GNU gettext en Windows?

Si estas usando Windows y necesitas instalar las utilidades GNU gettext para que django-admin makemessages funcione, consulta la sección "gettext en Windows" para obtener más información.

El formato de los archivos .po es sencillo. Cada archivo .po contiene una pequeña cantidad de metadatos tales como la información de contacto de quiénes mantienen la traducción, pero el grueso del archivo es una lista de *mensajes* – mapeos simples entre las cadenas de traducción y las traducciónes al idioma en cuestión propiamente dichas.

Por ejemplo, si tu aplicación Django contiene una cadena de traducción para el texto Bienvenido a mi sitio:

```
_("Bienvenido a mi sitio.")
```

Entonces el comando django-admin.py makemessages creara un archivo .po que contendrá el siguiente fragmento – un mensaje:

```
#: ruta/a/python/module.py:23
msgid "Bienvenido a mi sitio."
msgstr ""
```

Es necesaria una rápida explicación:

- **msgid** es la cadena de traducción, la cual aparece en el código fuente. No la modifiques.
- **msgstr** es donde colocas la traducción específica a un idioma. Su valor inicial es vacío de manera que es tu responsabilidad el cambiar esto. Asegúrate de mantener las comillas alrededor de tu traducción.

- Por conveniencia, cada mensaje incluye el nombre del archivo y el número de línea desde el cual la cadena de traducción fue extraída.

Los mensajes largos son un caso especial. La primera cadena inmediatamente a continuación de msgstr (o msgid) es una cadena vacía. El contenido en si mismo se encontrará en las próximas líneas con el formato de una cadena por línea. Dichas cadenas se concatenan en forma directa. ¡No olvides los espacios al final de las cadenas; en caso contrario todas serán agrupadas sin espacios entre las mismas!

Por ejemplo, a continuación vemos una traducción de múltiples líneas (extraída de la localización al español incluida con Django):

```
msgid ""
"There's been an error. It's been reported to the site administrators vía e-"
"mail and should be fixed shortly. Thanks for your patience."
msgstr ""
"Ha ocurrido un error. Se ha informado a los administradores del sitio "
"mediante correo electrónico y debería arreglarse en breve. Gracias por su "
"paciencia."
```

Observa los espacios finales.

TEN EN CUENTA EL CONJUNTO DE CARACTERES

Cuando crees un archivo .po con tu editor de texto favorito, primero edita la línea del conjunto de caracteres (busca por el texto "CHARSET") y fija su valor al del conjunto de caracteres usarás para editar el contenido. Generalmente, UTF-8 debería funcionar para la mayoría de los idiomas pero gettext debería poder manejar cualquier conjunto de caracteres.

Para reexaminar todo el código fuente y las plantillas en búsqueda de nuevas cadenas de traducción y actualizar todos los archivos de mensajes para *todos* los idiomas, ejecuta lo siguiente:

```
django-admin.py makemessages –a
```

Compilando archivos de mensajes

Luego de que has creado tu archivo de mensajes, y cada vez que realices cambios sobre el mismo necesitarás compilarlo a una forma más eficiente, según los usa gettext. Usa para ello la utilidad django-admin.py compilemessages.

Esta herramienta recorre todos los archivos .po disponibles y crea archivos .mo, los cuales son archivos binarios optimizados para su uso por parte de gettext. En el mismo directorio desde el cual ejecutaste django-admin.py makemessages, ejecuta django-admin.py compilemessages de la siguiente manera:

```
django-admin.py compilemessages
```

Y eso es todo. Tus traduccióones están listas para ser usadas.

Cómo descubre Django la preferencia de idioma

Una vez que has preparado tus traducciones – o, si solo deseas usar las que están incluidas en Django, necesitarás activar el sistema de traducción para tu aplicación. Detrás de escena, Django tiene un modelo muy flexible para decidir qué idioma se usará – determinado a nivel de la instalación, para un usuario particular, o ambas.

Para configurar una preferencia de idioma a nivel de la instalación, fija LANGUAGE_CODE en tu archivo de configuración. Django usará este idioma como la traducción por omisión – la opción a seleccionarse en último término si ningún otro traductor encuentra una traducción.

Si todo lo que deseas hacer es ejecutar Django con tu idioma nativo y hay disponible un archivo de idioma para el mismo, simplemente asigna un valor a LANGUAGE_CODE.

Si deseas permitir que cada usuario individual especifique el idioma que ella o él prefiere, usa LocaleMiddleware. LocaleMiddleware permite la selección del idioma basado en datos incluidos en la petición. Personaliza el contenido para cada usuario.

Para usar LocaleMiddleware, agrega django.middleware.locale.LocaleMiddleware a tu variable de configuración MIDDLEWARE_CLASSES. Debido a que el orden de los middlewares es relevante, deberías seguir las siguientes guías:

- Asegúrate de que se encuentre entre las primeras clases middleware instaladas.
- Debe estar ubicado después de SessionMiddleware, esto es debido a que LocaleMiddleware usa datos de la sesión.
- Si usas CacheMiddleware, coloca LocaleMiddleware después de este (de otra forma los usuarios podrían recibir contenido cacheado del locale equivocado).

  Por ejemplo tu MIDDLEWARE_CLASSES podría verse como esta:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
)
```

LocaleMiddleware intenta determinar la preferencia de idioma del usuario siguiendo el siguiente algoritmo:
Primero, busca una clave django_language en la sesión del usuario actual.

- Si eso falla, busca una cookie llamada django_language.
- Si eso falla, busca la cabecera HTTP Accept-Language. Esta cabecera es enviada por tu navegador y le indica al servidor qué idioma(s) prefieres en orden de prioridad. Django intenta con cada idioma que aparezca en dicha cabecera hasta que encuentra uno para el que haya disponible una traducción.
- Si eso falla, usa la variable de configuración global LANGUAGE_CODE.

En cada uno de dichas ubicaciones, el formato esperado para la preferencia de idioma es el formato estándar, como una cadena. Por ejemplo, portugués de Brasil es pt-br.

Si un idioma base está disponible pero el sub-idioma especificado no, Django usará el idioma base. Por ejemplo, si un usuario especifica de-at (alemán Austríaco) pero Django solo tiene disponible de, usará de.

Sólo pueden seleccionarse idiomas que se encuentren listados en la variable de configuración LANGUAGES. Si deseas restringir la selección de idiomas a un subconjunto de los idiomas provistos (debido a que tu aplicación no incluye todos esos idiomas), fija tu LANGUAGES a una lista de idiomas, por ejemplo:

```
LANGUAGES = (
   ('de', _('German')),
   ('en', _('English')),
)
```

Este ejemplo restringe los idiomas que se encuentran disponibles para su selección automática a alemán e inglés (y cualquier sub-idioma, como de-ch o en-us).

Si defines un LANGUAGES personalizado es posible marcar los idiomas como cadenas de traducción – pero usa una función gettext() "boba", no la que se encuentra en django.utils.translation. *Nunca* debes importar django.utils.translation desde el archivo de configuración debido a que ese módulo a su vez depende de las variables de configuración, y eso crearía una importación circular.

La solución es usar una función gettext()` "boba". A continuación un archivo de configuración de ejemplo:

```
ugettext = lambda s: s
LANGUAGES = (
   ('de', ugettext('German')),
   ('en', ugettext('English')),
)
```

Con este esquema, django-admin.py makemessages todavía será capaz de encontrar y marcar dichas cadenas para su traducción pero la misma no ocurrirá en tiempo de ejecución, de manera que tendrás que recordar envolver los idiomas con la *verdadera* gettext() en cualquier código que use LANGUAGES en tiempo de ejecución.

El LocaleMiddleware sólo puede seleccionar idiomas para los cuales exista una traducción base provista por Django. Si deseas ofrecer traducciónes para tu aplicación que no se encuentran en el conjunto de traducciónes incluidas en el código fuente de Django, querrás proveer al menos traducciónes básicas para ese idioma. Por ejemplo, Django usa identificadores de mensajes técnicos para traducir formatos de fechas y de horas – así que necesitarás al menos esas traducciónes para que el sistema funcione correctamente.

Un buen punto de partida es copiar el archivo .po de inglés y traducir al menos los mensajes técnicos, y quizá también los mensajes de los validadores.

Los identificadores de mensajes técnicos son fácilmente reconocibles; están completamente en mayúsculas. No necesitas traducir los identificadores de mensajes como lo haces con otros mensajes; en cambio, deber proporcionar la variante local correcta del valor provisto en inglés. Por ejemplo, con DATETIME_FORMAT (o DATE_FORMAT o TIME_FORMAT), este sería la cadena de formato que deseas usar en tu idioma. El formato es idéntico al de la cadena de formato usado por la etiqueta de plantillas now.

Una vez que el LocaleMiddleware ha determinado la preferencia del usuario, la deja disponible como request.LANGUAGE_CODE para cada objeto petición. Eres libre de leer este valor en tu código de vista. A continuación un ejemplo simple:

```
def hola_mundo(request):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Nota que con traducción estática (en otras palabras sin middleware) el idioma está en settings.LANGUAGE_CODE, mientras que con traducción dinámica (con middleware) el mismo está en request.LANGUAGE_CODE.

Usando traducciónes en tus propios proyectos

Django busca traducciónes siguiendo el siguiendo algoritmo:

- Primero, busca un directorio locale en el directorio de la aplicación correspondiente a la vista que se está llamando. Si encuentra una traducción para el idioma seleccionado, la misma será instalada.
- A continuación, busca un directorio locale en el directorio del proyecto. Si encuentra una traducción, la misma será instalada.
- Finalmente, verifica la traducción base en django/conf/locale.

De esta forma, puedes escribir aplicaciones que incluyan sus propias traducciónes, y puedes reemplazar traducciónes base colocando las tuyas propias en la ruta de tu proyecto. O puedes simplemente construir un proyecto grande a partir de varias aplicaciones y poner todas las traducciónes en un gran archivo de mensajes. Es tu elección.

¿Nota: Si estás configurando manualmente la variables de configuración, el directorio locale en el directorio del proyecto no será examinado, dado que Django pierde la capacidad de deducir la ubicación del directorio del proyecto. (Django normalmente usa la ubicación del archivo de configuración para determinar esto, y en el caso que estés configurado manualmente tus variables de configuración dicho archivo no existe).

Todos los repositorios de archivos de mensajes están estructurados de la misma forma:

- $APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)
- $PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)
- Todas las rutas listadas en LOCALE_PATHS en tu archivo de configuración son examinadas en el orden de búsqueda de <language>/LC_MESSAGES/django.(po|mo)
- $PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)

Para crear archivos de mensajes, usas la misma herramienta django-admin.py makemessages que usabas con los archivos de mensajes de Django. Solo necesitas estar en la ubicación adecuada – en el directorio en el cual exista ya sea el directorio conf/locale (en el caso del árbol de código fuente) o el directorio locale/ (en el caso de mensajes de aplicación o de proyecto). Usas también la misma herramienta django-admin.py compilemessages para producir los archivos binarios django.mo usados por gettext.

Tambien puedes ejecutar django-admin.py compilemessages --settings=path.to.settings Para hacer que el compilador procese todo los directorios de tu configuración LOCALE_PATHS.

Los archivos de mensajes de aplicaciones son un poquito complicados a la hora de buscar por los mismos – necesitas el LocaleMiddleware. Si no usas el middleware, solo serán procesados los archivos de mensajes de Django y del proyecto.

Finalmente, debes dedicarle tiempo al diseño de la estructura de tus archivos de traducción. Si tus aplicaciones necesitan ser enviadas a otros usuarios y serán usadas en otros proyectos, posiblemente quieras usar traducciónes específicas a dichas aplicaciones. Pero el usar traducciónes específicas a aplicaciones y aplicaciones en proyectos podrían producir problemas extraños con makemessages.py: makemessages recorrerá todos los directorios situados por debajo de la ruta actual y de esa forma podría colocar en el archivo de mensajes del proyecto identificadores de mensajes que ya se encuentran en los archivos de mensajes de la aplicación.

La salida más fácil de este problema es almacenar las aplicaciones que no son partes del proyecto (y por ende poseen sus propias traducciónes) fuera del árbol del proyecto. De esa forma django-admin.py makemessages ejecutado a nivel proyecto sólo traducirá cadenas que están conectadas a tu proyecto y no cadenas que son distribuidas en forma independiente.

La vista de redirección set_language

Por conveniencia, Django incluye una vista django.views.i18n.set_language, que fija la preferencia de idioma de un usuario y redirecciona de vuelta a la página previa.

Activa esta vista agregando la siguiente línea a tu URLconf:

```
url(r'^i18n/', include('django.conf.urls.i18n')),
```

Observa que este ejemplo hace que la vista esté disponible en /i18n/setlang/.
La vista espera ser llamada vía el método POST, con un parámetro language incluido en la cadena de petición. Si el soporte para sesiones está activo, la vista guarda la opción de idioma en la sesión del usuario. En caso contrario, guarda el idioma en una cookie django_language. (El nombre puede ser cambiado a través de la configuración LANGUAGE_COOKIE_NAME.)

Después de haber fijado la opción de idioma Django redirecciona al usuario, para eso sigue el siguiente algoritmo:

- Django busca un parámetro next en los datos POST.
- Si el mismo no existe o está vació, Django intenta la URL contenida en la cabecera Referer.
- Si la misma está vacía – por ejemplo, si el navegador de un usuario suprime dicha cabecera entonces el usuario será redireccionado a / (la raíz del sitio) como último recurso.

Este es un fragmento de código de plantilla HTML de ejemplo:

```
<form action="/i18n/setlang/" method="post">
<input name="next" type="hidden" value="/next/page/" />
<select name="language">
   {% for lang in LANGUAGES %}
      <option value="{{ lang.0 }}">{{ lang.1 }}</option>
   {% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

Traducciónes y JavaScript

Agregar traducciónes a JavaScript plantea algunos problemas:

- El código JavaScript no tiene acceso a una implementación de gettext.
- El código JavaScript no tiene acceso a los archivos .po o .mo; los mismos necesitan ser enviados desde el servidor.
- Los catálogos de traducción para JavaScript deben ser mantenidos tan pequeños como sea posible.

Django provee una solución integrada para estos problemas: convierte las traducciónes a JavaScript, de manera que puedas llamar a gettext y demás desde JavaScript.

La vista javascript_catalog

La solución principal a esos problemas es la vista javascript_catalog, que genera una biblioteca de código JavaScript con funciones que emulan la interfaz gettext más un arreglo de cadenas de traducción. Dichas cadenas de traducción se toman desde la aplicación, el proyecto o el núcleo de Django, de acuerdo a lo que especifiques ya sea en el diccionario info_dict o en la URL.

La forma de usar esto es así:

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns [
   url(r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
]
```

Cada cadena en package debe seguir la sintaxis de paquetes separados por puntos de Python (el mismo formato que las cadenas en INSTALLED_APPS) y deben referirse a un paquete que contenga un directorio locale. Si se especifican múltiples paquetes, todos esos catálogos son fusionados en un único catálogo. Esto es útil si usas JavaScript que usa cadenas de diferentes aplicaciones.

Puedes hacer que la vista sea dinámica colocando los paquetes en el patrón de la URL:

```
urlpatterns = [
    url(r'^jsi18n/(?P<packages>\S+?)/$, 'django.views.i18n.javascript_catalog'),
]
```

Con esto, especificas los paquetes como una lista de nombres de paquetes delimitados por un símbolo + en la URL. Esto es especialmente útil si tus páginas usan código de diferentes aplicaciones, este cambia frecuentemente y no deseas tener que descargar un único gran catálogo. Como una medida de seguridad, esos valores pueden solo tomar los valores django.conf o cualquier paquete de la variable de configuración INSTALLED_APPS.

Usando el catálogo de traducciónes JavaScript

Para usar el catálogo simplemente descarga el script generado dinámicamente de la siguiente forma:

```
<script type="text/javascript" src="/path/to/jsi18n/"></script>
```

Esta es la forma en la que el sitio de administración obtiene el catálogo de traducciónes desde el servidor. Una vez que se ha cargado el catálogo, tu código JavaScript puede usar la interfaz estándar gettext para acceder al mismo:

```
document.write(gettext('esto será traducido'));
```

Hay también un interfaz de ngettext:

```
var object_cnt = 1 // or 0, or 2, or 3, ...
s = ngettext('literal para algún caso en singular',
'literal para casos en plural', object_cnt);
```

E incluso, una función de interpolación de cadenas:

```
function interpolate(fmt, obj, named);
```

La sintaxis de interpolación se tomó prestada de Python, por lo que la función *interpolate* soporta tanto la interpolación por nombre y posicional.

- **Interpolación posicional**: un obj que contiene un arreglo de objetos cuyos valores de sus elementos son secuencialmente interpolados en su correspondiente marcador de posición fmt, en el mismo orden en que aparecen. Por ejemplo:

```
fmts = ngettext('There is %s object. Remaining: %s',
    'There are %s objects. Remaining: %s', 11);
s = interpolate(fmts, [11, 20]);
// s is 'There are 11 objects. Remaining: 20'
```

- **Interpolación por nombre**: Este modo es seleccionado pasando opcionalmente un parámetro booleano *named* como true. El obj puede contener un objeto JavaScript o un arreglo asociado. Por ejemplo:

```
d = {
    count: 10
    total: 50
```

```
};

fmts = ngettext('Total: %(total)s, there is %(count)s object',
    'there are %(count)s of a total of %(total)s objects', d.count);
s = interpolate(fmts, d, true);
```

Sin embargo, no debes exagerar con el uso de la interpolación de cadenas – esto sigue siendo JavaScript así que el código tendrá que realizar múltiples sustituciones de expresiones regulares. Esto no es tan rápido como la interpolación de cadenas en Python, de manera que deberías reservarlo para los casos en los que realmente lo necesites (por ejemplo en combinación con ngettext para generar pluralizaciones en forma correcta).

Creando catálogos de traducciónes JavaScript

Los catálogos de traducciónes se crean y actualizan de la misma manera que el resto de los catálogos de traducciónes de Django, con la herramienta django-admin.py makemessages. La única diferencia es que es necesario que proveas un parámetro -d djangojs, de la siguiente forma:

```
django-admin.py makemessages -d djangojs -l de
```

Esto crea o actualiza el catálogo de traducción para JavaScript para alemán. Luego de haber actualizado catálogos, sólo ejecuta django-admin.py compilemessages de la misma manera que lo haces con los catálogos de traducción normales de Django.

Notas para usuarios familiarizados con gettext

Si conoces gettext podrías notar las siguientes particularidades en la forma en que Django maneja las traducciónes:

- El dominio de las cadenas es django o djangojs. El dominio de cadenas se usa para diferenciar entre diferentes programas que almacenan sus datos en una biblioteca común de archivos de mensajes (usualmente /usr/share/locale/). EL dominio django se usa para cadenas de traducción de Python y plantillas y se carga en los catálogos de traducciónes globales. El dominio djangojs se usa sólo para catálogos de traducciónes de JavaScript para asegurar que los mismos sean tan pequeños como sea posible.

- Django sólo usa gettext y gettext_noop. Esto es debido a que Django siempre usa internamente cadenas DEFAULT_CHARSET. Usar ugettext no significaría muchas ventajas ya que de todas formas siempre necesitarás producir UTF-8.

- Django no usa xgettext en forma independiente. Usa envoltorios Python alrededor de xgettext y msgfmt. Esto es más que nada por conveniencia.

gettext en Windows

La siguiente sección es únicamente para personas que necesitan extraer mensajes IDs o compilar archivos de mensajes (.po) en Windows. El trabajo de traducción en sí mismo, sólo implica editar los archivos existentes de este tipo, excepto si quieres

crear tus propios archivos de mensajes, o si deseas compilar o probar cambios en archivos de mensajes, necesitaras usar las utilidades gettext.

Descarga los siguientes archivos zip de los servidores de GNOME desde: http://ftp.gnome.org/pub/gnome/binaries/win32/dependencies/ o de cualquiera de los espejos: http://ftp.gnome.org/pub/GNOME/MIRRORS

- gettext-runtime-x.zip
- gettext-tools-x.zip

Donde x es el número de versión, la mínima requerida es la 0.15.

Extrae los 3 archivos en un mismo folder (por ejemplo: C:\Archivos de Programas\utilidades gettext)

Actualiza las rutas del sistema (PATH):

- Dirígete al Panel de Control > Sistema > Avanzados > Variables de entorno
- En la lista de Variables del sistema busca Path, da clic en Editar
- Y agrega; C:\Archivos de Programas\utilidades gettext\bin al final del valor del campo valor de la variable.

Puedes usar también los binarios gettext obtenidos en alguna otra parte, usando la versión larga xgettext --version para que el comando trabaje adecuadamente.

No intentes usar en Django, utilidades de traducción con un paquete gettext si el comando xgettext --version causa errores al ejecutarse en una ventana de una terminal tal como “xgettext.exe has generated errors and will be closed by Windows”. Algunos binarios de la serie 0.14.4 no soportan este comando.

¿Qué sigue?

El *capítulo final* se enfoca en la seguridad – como proteger tu sitio y a tus usuarios de atacantes maliciosos.

CAPÍTULO 20

Seguridad

"Internet puede ser un lugar aterrador"

En estos tiempos, los papelones de seguridad en internet con alta exposición pública parecen ser cosa de todos los días. Hemos visto virus propagarse con una velocidad asombrosa, ejércitos de computadoras comprometidas ser empuñadas como armas, una interminable *carrera armamentista* contra los *spammers*, y muchos, muchos reportes de robos de identidad de sitios Web *hackeados*.

Parte de las tareas de un desarrollador Web es hacer lo que esté en sus manos para combatir esas fuerzas de la oscuridad. Todo desarrollador Web necesita considerar la seguridad como un aspecto fundamental de la programación Web. Desafortunadamente, se da el caso de que implementar la seguridad es *difícil* – los atacantes sólo necesitan encontrar una única vulnerabilidad, pero los defensores deben proteger todas y cada una.

Django intenta mitigar esta dificultad. Está diseñado para protegerte automáticamente de muchos de los errores de seguridad comunes que cometen los nuevos (e incluso los experimentados) desarrolladores Web. Aun así, es importante entender de qué se tratan dichos problemas, cómo es que Django te protege, y – esto es lo más importante los pasos que puedes tomar para hacer tu código aun más seguro.

Antes, sin embargo, una importante aclaración: No es nuestra intención presentar una guía definitiva sobre todos los *exploits* de seguridad Web conocidos, y tampoco trataremos de explicar cada vulnerabilidad en una forma completa. En cambio, presentaremos una breve sinopsis de problemas de seguridad que son relevantes para Django.

El tema de la seguridad en la Web

Si aprendes sólo una cosa de este capítulo, que sea esta:

Nunca -- bajo ninguna circunstancia -- confíes en datos enviados por un navegador.

Nunca sabes quién está del otro lado de esa conexión HTTP. Podría tratarse de uno de tus usuarios, pero de igual forma podría tratarse de un vil cracker buscando el más mínimo resquicio.

Cualquier dato de cualquier naturaleza que arriba desde el navegador necesita ser tratado con una generosa dosis de paranoia. Esto incluye tanto datos que se encuentran *"in band"* (por ejemplo enviados desde formularios Web) como *"out of band"* (por ejemplo cabeceras HTTP, cookies, y otra información de petición). Es trivial falsificar los metadatos de la petición que los navegadores usualmente agregan automáticamente.

Todas las vulnerabilidades tratadas en este capítulo derivan directamente de confiar en datos que arriban a través del cable y luego fallar a la hora de limpiar esos datos antes de usarlos. Debes convertir en una práctica general el preguntarte "¿De dónde vienen estos datos?"

Inyección de SQL

La *inyección de SQL* es un exploit común en el cual un atacante altera los parámetros de la página (tales como datos de GET/POST o URLs) para insertar fragmentos arbitrarios de SQL que una aplicación Web ingenua ejecuta directamente en su base de datos. Es probablemente la más peligrosa y, desafortunadamente una de las más comunes vulnerabilidad existente.

Esta vulnerabilidad se presenta más comúnmente cuando se está construyendo SQL "a mano" a partir de datos ingresados por el usuario. Por ejemplo, imaginemos que se escribe una función para obtener una lista de información de contacto desde una página de búsqueda. Para prevenir que los spammers lean todas las direcciones de email en nuestro sistema, vamos a exigir al usuario que escriba el nombre de usuario del cual quiere conocer sus datos antes de proveerle la dirección de email respectiva:

```
def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = '%s';" % username
    # ejecuta consultas SQL aqui...
```

■**Nota:** En este ejemplo, y en todos los ejemplos similares del tipo "no hagas esto" que siguen, hemos omitido deliberadamente la mayor parte del código necesario para hacer que el mismo realmente funcione. No queremos que este código sirva, si accidentalmente alguien lo toma fuera de contexto y lo usa.

A pesar de que a primera vista eso no parece peligroso, realmente lo es.

Primero, nuestro intento de proteger nuestra lista de emails completa va a fallar con una consulta construida en forma ingeniosa. Pensemos acerca de qué sucede si un atacante escribe "'OR' 'a'='a" en la caja de búsqueda. En ese caso, la consulta que la interpolación construirá será:

```
SELECT * FROM user_contacts WHERE username = '' OR 'a' = 'a';
```

Debido a que hemos permitido SQL sin protección en la string, la cláusula OR agregada por el atacante logra que se retornen todos los registros.

Sin embargo, ese es el *menos* pavoroso de los ataques. Imaginemos qué sucedería si el atacante envía "'; DELETE FROM user_contacts WHERE 'a' = 'a'". Nos encontraríamos con la siguiente consulta completa:

```
SELECT * FROM user_contacts WHERE username = '';
 DELETE  FROM user_contacts  WHERE 'a' = 'a';
```

¡Ouch! ¿Donde iría a parar nuestra lista de contactos?

La solución

Aunque este problema es insidioso y a veces difícil de detectar la solución es simple: *nunca* confíes en datos provistos por el usuario y *siempre escapa* el mismo cuando lo conviertes en SQL.

La API de base de datos de Django hace esto por ti. Escapa automáticamente todos los parámetros especiales SQL, de acuerdo a las convenciones de uso de comillas del servidor de base de datos que estés usando (por ejemplo, PostgreSQL o MySQL).

Por ejemplo, en esta llamada a la API:

```
foo.get_list(bar__exact="' OR 1=1")
```

Django *escapará* la entrada apropiadamente, resultando en una sentencia como esta:

```
SELECT * FROM foo WHERE bar = '\' OR 1=1'
```

Que es completamente inocua.

Esto se aplica a la totalidad de la API de base de datos de Django, con un par de excepciones:

El argumento where del método extra() (ver Apéndice C). Dicho parámetro acepta, por diseño, SQL crudo.

Consultas realizadas "a mano" usando la API de base de datos de nivel más bajo.

En tales casos, es fácil mantenerse protegido. para ello evita realizar interpolación de strings y en cambio usa *parámetros asociados* (*bind parameters*). Esto es, el ejemplo con el que comenzamos esta sección debe ser escrito de la siguiente manera:

```
from django.db import connection

def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = %s;"
    cursor = connection.cursor()
    cursor.execute(sql, [user])
    # ... haz algo cono los resultados
```

El método de bajo nivel execute toma un string SQL con marcadores de posición %s y automáticamente *escapa* e inserta parámetros desde la lista que se le provee como segundo argumento. Cuando construyas SQL en forma manual hazlo *siempre* de esta manera.

Desafortunadamente, no puedes usar *parámetros asociados* en todas partes en SQL; no son permitidos como identificadores (esto es, nombres de tablas o columnas). Así que, si, por ejemplo, necesitas construir dinámicamente una lista de tablas a partir de una variable enviada mediante POST, necesitarás *escapar* ese nombre en tu código. Django provee una función, django.db.backend.quote_name, la cual *escapará* el identificador de acuerdo al esquema de uso de comillas de la base de datos actual.

Cross-Site Scripting

El *Cross-site scripting* (XSS) (Scripting inter-sitio), puede encontrarse en aplicaciones Web que fallan a la hora de *escapar* en forma correcta contenido provisto por el usuario antes de renderizarlo en HTML. Esto le permite a un atacante insertar HTML arbitrario en tu página Web, usualmente en la forma de etiquetas <script>.

Los atacantes a menudo usan ataques XSS para robar información de cookies y sesiones, o para engañar usuarios y lograr que proporcionen información privada a la persona equivocada (también conocido como *phishing*).

Este tipo de ataque puede tomar diferentes formas y tiene prácticamente infinitas permutaciones, así que sólo vamos a analizar un ejemplo típico. Consideremos esta simple vista "Hola mundo":

```
from django.http import HttpResponse

def di_hola(request):
   nombre = request.GET.get('nombre')
   return HttpResponse('<h1>Hola, %s!</h1>' % nombre)
```

Esta vista simplemente lee un nombre desde un parámetro GET y pasa dicho nombre a la plantilla. Podríamos escribir una plantilla para esta vista de la siguiente manera:

```
<h1>Hola, {{ nombre }}!</h1>
```

De manera que si accediéramos a http://example.com/hola/?name=Jacob, la página renderizada contendría lo siguiente:

```
<h1>Hola, Jacob!</h1>
```

Pero atención – ¿qué sucede si accedemos a http://example.com/hello/?nombre=Jacob? En ese caso obtenemos esto:

```
<h1>Hola, <i>Jacob</i>!</h1>
```

Obviamente, un atacante no usará algo tan inofensivo como una etiquetas ; podría incluir un fragmento completo de HTML que se apropiara de tu página insertando contenido arbitrario. Este tipo de ataques ha sido usado para engañar a usuarios e inducirlos a introducir datos en lo que parece ser el sitio Web de su banco, pero en efecto es un formulario saboteado vía XSS que envía su información bancaria a un atacante.

El problema se complica aún más si almacenas estos datos en la base de datos y luego la visualizas en tu sitio. Por ejemplo, en una oportunidad se encontró que MySpace era vulnerable a un ataque XSS de esta naturaleza. Un usuario había insertado JavaScript en su página de perfil, dicho código lo agregaba a la lista de amigos de todos los usuarios que visitaran su página de perfil. En unos pocos días llegó a tener millones de amigos.

Ahora, esto podría sonar relativamente inofensivo, pero no olvides que este atacante había logrado que *su* código – no el código de MySpace – se ejecutara en *tu* computadora. Esto viola la confianza asumida acerca de que todo el código ubicado en MySpace es realmente escrito por MySpace.

MySpace fue muy afortunado de que este código malicioso no hiciera cosas como borrar automáticamente las cuentas de los usuarios que lo ejecutaran, o cambiar sus contraseñas, o inundar el sitio con spam, o cualquiera de los otros escenarios de pesadilla que esta vulnerabilidad hace posibles.

La solución

La solución es simple: *siempre* escapa *todo* el contenido que pudiera haber sido enviado por un usuario, antes de insertarlo dentro del HTML.

Para protegerte contra esto, el sistema de la plantilla de Django automáticamente escapa todos los valores de las variables. Veamos qué sucede si reescribimos nuestro ejemplo usando el sistema de plantillas:

```
# views.py
from django.shortcuts import render

def di_hola(request):
    nombre = request.GET.get('nombre', 'mundo')
    return render(request, 'hola.html', {'nombre': nombre})
```

```
# hola.html
<h1>Hola, {{ nombre }}!</h1>
```

Con esto en su lugar Django automáticamente escapa los valores de las variables, en este caso la variable nombre. Sin embargo aun podemos manualmente usar un filtro escape en la plantilla:

```
<h1>Hola, {{ nombre|escape }}!</h1>
```

De esta manera ya no seríamos vulnerables. Debes usar *siempre* la etiqueta escape (o algo equivalente) cuando visualizas en tu sitio contenido enviado por el usuario.

En él *capítulo 4* tratamos el tema del auto-escape y la forma de desactivarlo. Por lo que siempre que uses esta característica, asegurarte de activarlo. Aun si Django incorpora esta característica *debes* formarte el hábito de preguntarte, en todo momento, "¿De dónde provienen estos datos?". Ya que ninguna solución automática protegerá tu sitio de ataques XSS el 100% del tiempo.

Cross-Site Request Forgery

Los ataques mediante cross-site request forgery (CSRF) (Falsificación de peticiones inter-sitio) suceden, cuando un sitio Web malicioso engaña a los usuarios y los induce a visitar una URL desde un sitio ante el cual ya se han autenticado – por lo tanto saca provecho de su condición de usuario ya autenticado.

Django incluye herramientas para proteger ante este tipo de ataques. Tanto el ataque en sí mismo como dichas herramientas son tratados con gran detalle en el *capítulo 16*.

Session Forging/Hijacking

No se trata de un ataque específico, sino una clase general de ataques sobre los datos de sesión de un usuario. Puede tomar diferentes formas:

- Un ataque del tipo *man-in-the-middle*, en el cual un atacante espía datos de sesión mientras estos viajan por la red (cableada o inalámbrica).

- *Session forging* (Falsificación de sesión), en la cual un atacante usa un identificador de sesión (posiblemente obtenido mediante un ataque man-in-the-middle) para simular ser otro usuario.

  Un ejemplo de los dos primeros sería una atacante en una cafetería usando la red inalámbrica del lugar para capturar una cookie de sesión. Podría usar esa cookie para hacerse pasar por el usuario original.

- Un ataque de *falsificación de cookies* en el cual un atacante sobrescribe los datos almacenados en una cookie que en teoría no son modificables. Él *capítulo 14* explica en detalle cómo funcionan las cookies, y uno de los puntos salientes es que es trivial para los navegadores y usuarios maliciosos el cambiar las cookies sin tu conocimiento.

  Existe una larga historia de sitios Web que han almacenado una cookie del tipo IsLoggedIn=1 o aun LoggedInAsUser=jacob. Es trivialmente sencillo sacar provecho de ese tipo de cookies.

  En un nivel aun más sutil, nunca será una buena idea confiar en nada que se almacene en cookies; nunca sabes quién puede haber estado manoseando las mismas.

- *Session fixation* (fijación de sesión), en la cual un atacante engaña a un usuario y logra asignar un nuevo valor o limpiar el valor existente del identificador de su sesión.

  Por ejemplo, PHP permite que los identificadores de sesión se pasen en la URL (por ejemplo, http://example.com/?PHPSESSID=fa90197ca25f6ab40bb1374c510d7a32). Un atacante que logre engañar a un usuario para que haga clic en un link que posea un identificador de sesión fijo causará que ese usuario comience a usar esa sesión.

  La fijación de sesión se ha usado en ataques de *phishing* para engañar a usuarios e inducirlos a ingresar información personal en una cuenta que está bajo el control de atacante. Este puede luego conectarse al sitio con dicho usuario y obtener datos.

- *Session poisoning* (envenenamiento de sesión), en el cual in atacante inyecta datos potencialmente peligrosos en la sesión de un usuario – usualmente a través de un formulario que el usuario envía con datos de su sesión.

  Un ejemplo canónico es un sitio que almacena un valor de preferencia simple (como el color de fondo de una página) en una cookie. Un atacante podría engañar a un usuario e inducirlo a hacer clic en un link que envía un "color" que en realidad contiene un ataque XSS; si dicho color no está siendo *escapado*, el usuario podría insertar nuevamente código malicioso en el entorno del usuario.

La solución

Existe un número de principios generales que pueden protegerte de estos ataques:

- Nunca permitas que exista información sobre sesiones contenida en las URLs.
  - El framework de sesiones de Django (ver *capítulo 14*) simplemente no permite que las URLs contengan sesiones.

- No almacenes datos en cookies en forma directa; en cambio, almacena un identificador de sesión que esté relacionado a datos de sesión almacenados en el back-end.

- Si usas el framework de sesiones incluido en Django (o sea request.session), eso es manejado en forma automática. La única cookie que usa el framework de sesiones es un identificador de sesión; todos los datos de la sesiones se almacenan en la base de datos.

- Recuerda *escapar* los datos de la sesión si los visualizas en la plantilla. Revisa la sección previa sobre XSS y recuerda que esto se aplica a cualquier contenido creado por el usuario así como a cualquier dato enviado por el navegador. Debes considerar la información de sesiones como datos creados por el usuario.

Previene la falsificación de de identificadores de sesión por parte de un atacante siempre que sea posible.

A pesar de que es prácticamente imposible detectar a alguien que se ha apropiado de un identificador de sesión, Django incluye protección contra un ataque de sesiones de fuerza bruta.

Los identificadores de sesión se almacenan como hashes (en vez de números secuenciales) lo que previene un ataque por fuerza bruta, y un usuario siempre obtendrá un nuevo identificador de sesión si intenta usar uno no existente, lo que previene la *session fixation*.

Nota que ninguno de estos principios y herramientas previene ante ataques man-in-the-middle. Dichos tipos de ataques son prácticamente imposibles de detectar. Si tu sitio permite que usuarios identificados visualicen algún tipo de datos importantes debes, *siempre*, publicar dicho sitio vía HTTPS. Adicionalmente, si tienes un sitio con SSL, debes asignar a la variable de configuración SESSION_COOKIE_SECURE el valor True; esto hará que Django envíe las cookies de sesión vía HTTPS.

Inyección de cabeceras de email

La hermana menos conocida de la inyección de SQL, la *inyección de cabeceras de email*, toma control de formularios Web que envían emails. Un atacante puede usar esta técnica para enviar spam mediante tu servidor de email. Cualquier formulario que construya cabeceras de email a partir de datos de un formulario Web es vulnerable a este tipo de ataque.

Analicemos el formulario de contacto canónico que puede encontrarse en muchos sitios. Usualmente el mismo envía un mensaje a una dirección de email fija y, por lo tanto, a primera vista no parece ser vulnerable a abusos de spam.

Sin embargo, muchos de esos formularios permiten también que los usuarios escriban su propio asunto para el email (en conjunto con una dirección "de", el cuerpo del mensaje y a veces algunos otros campos). Este campo asunto es usado para construir la cabecera "subject" del mensaje de email.

Si dicha cabecera no es *escapada* cuando se construye el mensaje de email, un atacante podría enviar algo como "hello\ncc:spamvictim@example.com" (donde "\n" es un carácter de salto de línea). Eso haría que las cabeceras de email fueran:

```
To: hardcoded@example.com
Subject: hello
cc: spamvictim@example.com
```

Como en la inyección de SQL, si confiamos en la línea de asunto enviada por el usuario, estaremos permitiéndole construir un conjunto malicioso de cabeceras, y podrá usar nuestro formulario de contacto para enviar spam.

La solución

Podemos prevenir este ataque de la misma manera en la que prevenimos la inyección de SQL: *escapando* o verificando siempre el contenido enviado por el usuario.

Las funciones de mail incluidas en Django (en django.core.mail) simplemente no permiten saltos de línea en ninguno de los campos usados para construir cabeceras (las direcciones de y para, más el asunto). Si intentas usar django.core.mail.send_mail con un asunto que contenga saltos de línea, Django arrojará una excepción BadHeaderError.

Si no usas las funciones de email de Django para enviar email, necesitarás asegurarte de que los saltos de línea en las cabeceras o causan un error o son eliminados. Podrías querer examinar la clase SafeMIMEText en django.core.mail para ver cómo implementa esto Django.

Directorio Transversal

Directorio Transversal se trata de otro ataque del tipo inyección, en el cual un usuario malicioso subvierte código de manejo de sistema de archivos para que lea y/o escriba archivos a los cuales el servidor Web no debería tener acceso.

Un ejemplo podría ser una vista que lee archivos desde el disco sin limpiar cuidadosamente el nombre de archivo:

```
def dump_file(request):
    filename = request.GET["filename"]
    filename = os.path.join(BASE_PATH, filename)
    content = open(filename).read()
    # ...
```

A pesar que parece que la vista restringe el acceso a archivos que se encuentren más allá que BASE_PATH (usando os.path.join), si el atacante envía un filename que contenga .. (esto es, dos puntos, una notación corta para “el directorio padre”), podría acceder a archivos que se encuentren “más arriba” que BASE_PATH. De allí en más es sólo una cuestión de tiempo el hecho que descubra el número correcto de puntos para acceder exitosamente, por ejemplo a ../../../../../etc/passwd.

Todo aquello que lea archivos sin el *escapado* adecuado es vulnerable a este problema. Las vistas que *escriben* archivos son igual de vulnerables, pero las consecuencias son doblemente calamitosas.

Otra permutación de este problema yace en código que carga módulos dinámicamente a partir de la URL u otra información de la petición. Un muy público ejemplo se presentó en el mundo de Ruby on Rails. Con anterioridad a mediados del 2006, Rails usaba URLs como http://example.com/person/poke/1 directamente para cargar módulos e invocar métodos. El resultado fué que una URL cuidadosamente construida podía cargar automáticamente código arbitrario, ¡incluso un script de reset de base de datos!

La solución

Si tu código necesita alguna vez leer o escribir archivos a partir de datos ingresados por el usuario, necesitas limpiar muy cuidadosamente la ruta solicitada para asegurarte que un atacante no pueda escapar del directorio base más allá del cual estás restringiendo el acceso.

¡Nunca debes escribir código que pueda leer cualquier área del disco! Un buen ejemplo de cómo hacer este *escapado* yace en la vista de publicación de contenidos estáticos (en django.view.static).

Este es el código relevante:

```
import os
import posixpath

# ...

    path = posixpath.normpath(urllib.unquote(path))
    newpath = ''
    for part in path.split('/'):
        if not part:
        # strip empty path components
            continue

     drive, part = os.path.splitdrive(part)
     head, part = os.path.split(part)
     if part in (os.curdir, os.pardir):
         # strip '.' and '..' in path
        continue

     newpath = os.path.join(newpath, part).replace('\\', '/')
```

Django no lee archivos (a menos que uses la función static.serve, pero en ese caso está protegida por el código recién mostrado), así que esta vulnerabilidad no afecta demasiado el código del núcleo.

Adicionalmente, el uso de la abstracción de URLconf significa que Django *solo* cargará código que le hayas indicado explícitamente que cargue. No existe manera de crear una URL que cause que Django cargue algo no mencionado en una URLconf.

Exposición de mensajes de error

Mientras se desarrolla, tener la posibilidad de ver tracebacks y errores en vivo en tu navegador es extremadamente útil. Django posee mensajes de depuración "vistosos" e informativos específicamente para hacer la tarea de depuración más fácil.

Sin embargo, si esos errores son visualizados una vez que el sitio pasa a producción, pueden revelar aspectos de tu código o configuración que podrían ser de utilidad a un atacante.

Es más, los errores y tracebacks no son para nada útiles para los usuarios finales. La filosofía de Django es que los visitantes al sitio nunca deben ver mensajes de error relacionados a una aplicación. Si tu código genera una excepción no tratada, un visitante al sitio no debería ver un traceback completo – ni *ninguna* pista de fragmentos de código o mensajes de error (destinados a programadores) de Python. En cambio, el visitante debería ver un amistoso mensaje "Esta página no está disponible".

Naturalmente, por supuesto, los desarrolladores necesitan ver tracebacks para depurar problemas en su código. Así que el framework debería ocultar todos los mensajes de error al público pero debería mostrarlos a los desarrolladores del sitio.

La solución

Como se mostro en el *capítulo 12*, Django contiene un sencillo control que gobierna la visualización de esos mensajes de error. Si se fija la variable de configuración DEBUG al valor True, los mensajes de error serán visualizados en el navegador. De otra forma, Django retornará un mensaje HTTP 500 ("Error interno del servidor") y renderizará una plantilla de error provista por ti. Esta plantilla de error tiene el nombre 500.html y debe estar situada en la raíz de uno de tus directorios de plantillas.

Dado que los desarrolladores aun necesitan ver los errores que se generan en un sitio en producción, todos los errores que se manejen de esta manera dispararán el envío de un email con el traceback completo a las direcciones de correo configuradas en la variable ADMINS.

Los usuarios que implementen en conjunto con Apache y wsgi_python deben también asegurarse que tienen PythonDebug Off en sus archivos de configuración de Apache; esto suprimirá cualquier error que pudiera ocurrir aun antes de que Django se haya cargado.

Unas palabras finales sobre la seguridad

Esperamos que esta pequeña exposición sobre problemas de seguridad no sea demasiado intimidante. Es cierto que la Web puede ser un mundo salvaje y confuso, pero con un poco de previsión puedes tener un sitio Web seguro.

Ten en mente que la seguridad Web es un campo en constante cambio; si estás leyendo la versión en papel de este libro, asegúrate de consultar recursos sobre seguridad más actuales en búsqueda de nuevas vulnerabilidades que pudieran haber sido descubiertas recientemente. En efecto, siempre es una buena idea dedicar algún tiempo semanalmente o mensualmente a investigar y mantenerte actualizado acerca del estado de la seguridad de aplicaciones Web. Es una pequeña inversión a realizar, pero la protección que obtendrás para ti y tus usuarios no tiene precio.

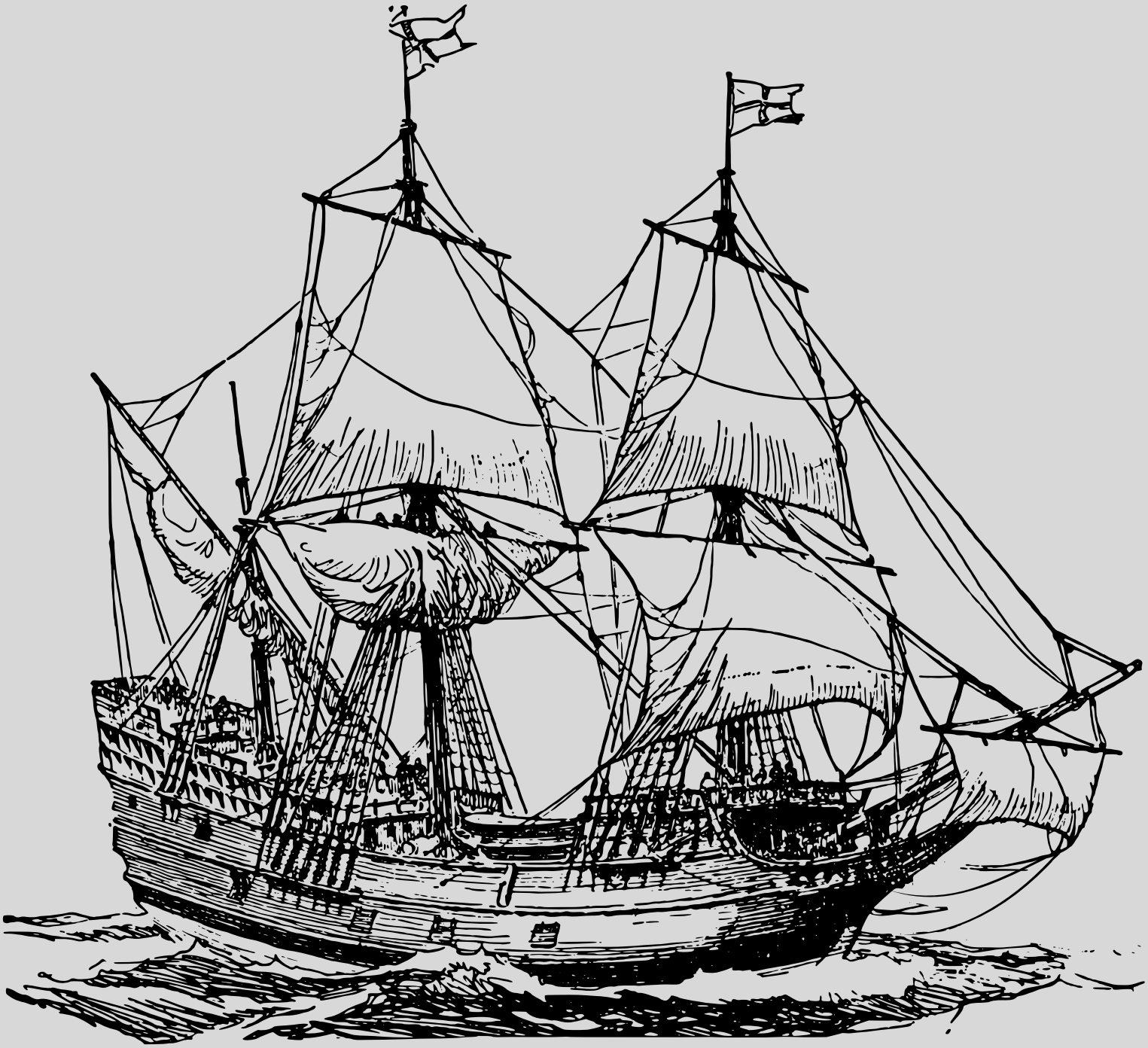
¿Qué sigue?

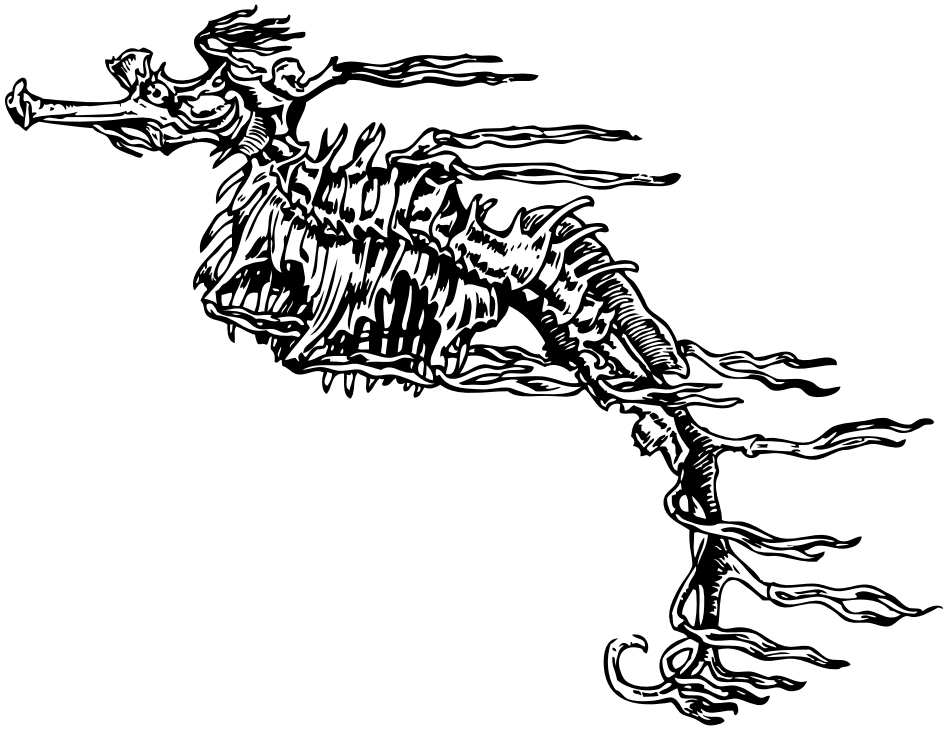
Has alcanzado el final de nuestro programa regular. Los siguientes apéndices contienen material de referencia que puedes necesitar a medida que trabajes sobre tus proyectos Django

Te deseamos la mejor de las suertes en la puesta en marcha de tu sitio Django, ya sea un pequeño juguete para ti y tus amigos o el próximo gran invento de internet ☺.

PARTE 4

Apéndice





APÉNDICE A

Referencia de la definición de modelos

El *capítulo 5* explica lo básico sobre la definición de modelos, y lo utilizamos a lo largo del libro, sin embargo existe un enorme rango de opciones disponibles que no se han cubierto. Este apéndice explica toda opción disponible en la definición de modelos.

A pesar de que estas APIs se consideran muy estables, los desarrolladores de Django agregan en forma consistente y constante nuevos atajos y conveniencias a la definición de modelos, por lo que es buena idea comprobar siempre la documentación más reciente en línea, disponible en la página del proyecto:
http://www.djangoproject.com/documentation/

Campos

Un campo es una clase abstracta que representa una columna de una tabla de una base de datos. Django usa campos o fields para crear las tablas de la base de datos.

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

Restricciones en el nombre de los campos

Django pone solo dos restricciones en el nombre de los campos:

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

```
class Ejemplo(models.Model):
    pass = models.IntegerField() #¡ 'pass' es una palabra reservada en Python!
```

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

```
class Ejemplo(models.Model):
    foo__bar = models.IntegerField() # 'foo__bar' tiene dos guiones bajos!
```

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

Las palabras reservadas de SQL, como join, where, o select, *son* permitidas como nombres de campo, dado que Django "escapa" todos los nombres de tabla y columna de la base de datos en cada consulta SQL subyacente. Utiliza la sintaxis de "comillas" del motor de base de datos particular.

Cada campo en tu modelo debe ser una instancia de la clase de campo apropiada. Django usa los tipos de clase Field para determinar algunas cosas:
El tipo de columna de la base de datos (ej.: INTEGER, VARCHAR).

El widget a usar en la interfaz de administración de Django, si vas a usarla (ej., <input type="text">, <select>).

Los requerimientos mínimos de validación, que se usan en la interfaz de administración de Django.

A continuación, una lista completa de las clases de campo, ordenadas alfabéticamente. Los campos de relación (ForeignKey, etc.) se tratan en la siguiente sección.

AutoField

Un IntegerField que se incrementa automáticamente de acuerdo con los IDs disponibles. Normalmente no necesitarás utilizarlos directamente; se agrega un campo de clave primaria automáticamente a tu modelo si no especificas una clave primaria.

BigintegerField

Un campo para almacenar números enteros de 64 bits, es parecido a un campo IntegerField excepto que garantiza ajustar números de 9223372036854775808 a 9223372036854775807.

BinaryField

Un campo que almacena datos binarios. Únicamente soporta la asignación de bytes. Ten en cuenta que este campo tiene funcionalidades limitadas. Por ejemplo no es posible filtrar un queryset en un valor de un campo BinaryField.

BooleanField

Un campo Verdadero/Falso.

PARA USUARIOS DE MYSQL

Un campo booleano en MySQL es almacenado en una columna TINYINT con un valor de 0 o 1 (la mayoría de las bases de datos tienen una apropiada instancia de tipo BOLEAN.) Por lo que en MySQL únicamente cuando se recupera un valor almacenado BooleanField de la base de datos, este tiene un valor de 1 o 0, en lugar de True o False. Normalmente esto no debería ser un problema, ya que Python garantiza que 1 == True y 0 == False. Simplemente se cuidadoso si escribes algo como: obj is True cuando obj es el valor de un atributo boleano de un modelo. Si el modelo fue construido usando el backend mysql, las pruebas "is" fallaran. Es mejor usar una prueba usando "==" en casos como estos.

CharField

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

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

CommaSeparatedIntegerField

Un campo de enteros separados por comas. Igual que en CharField, se requiere el argumento max_length.

DateField

Un campo de fecha. DateField tiene algunos argumentos opcionales extra, como se muestra en la Tabla A-1.

Argumento	Descripción
auto_now	Asigna automáticamente al campo un valor igual al momento en que se guarda el objeto. Es útil para las marcas de tiempo "última modificación". Observar que *siempre* se usa la fecha actual; no es un valor por omisión que se pueda sobrescribir.
auto_now_add	Asigna automáticamente al campo un valor igual al momento en que se crea el objeto. Es útil para la creación de marcas de tiempo. Observar que *siempre* se usa la fecha actual; no es un valor por omisión que se pueda sobreescribir.

Tabla A1: *Argumentos opcionales extra de DateField*

DateTimeField

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

DecimalField

Un numero decimal de precisión-fija, representado en Python por una instancia de decimal. Decimal. Requiere de dos argumentos:

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

Tabla A2: *Argumentos opcionales extra de DateField*

Por ejemplo, para almacenar números hasta 999 con una resolución de dos decimales, usa:

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

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

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

Cuando asignes a DecimalField, utiliza cualquiera de los objetos decimal.Decimal o una cadena – no un numero de punto flotante Python.

EmailField

Un CharField que chequea que el valor sea una dirección de email válida. No acepta max_length; max_length se establece automáticamente en 254.

FileField

Un campo para subir archivos.

■**Nota:** Los argumentos primary_key y unique, no están soportados, si los usas se lanzara un error del tipo: TypeError

---

Requiere de dos argumentos opcionales:

upload_to: La ruta del sistema de archivos local que se agregará a la configuración de MEDIA_ROOT para determinar el valor de el atributo django.core.files.File.url.

La ruta puede contener el "formato *strftime*", (consulta la documentación de Python para obtener ayuda sobre el modulo estándar *time*) el cual se usa para remplazar la fecha/tiempo de el archivo a subir (de modo que los ficheros subidos no llenen el directorio dado.)

También puede contener un llamable, tal como una función, la cual debe ser llamada para obtener la ruta a la cual subir los archivos, incluyendo el nombre del archivo. Este llamable debe permitir aceptar dos argumentos y devolver la ruta estilo tipo Unix (con barras inclinadas) para pasarlas a el sistema de almacenamiento. Los dos argumentos que puedes pasarle son:

Argumento	Descripción
instance	Una instancia de un modelo donde este definido el campo FileField. Específicamente, esta es la instancia en particular en donde el fichero actual esta adjunto. En la mayoría de los casos, este objeto no habrá sido guardado en la base de datos todavía, así que si utilizas por defecto AutoField, *puede ser que todavía no tenga un valor para el campo de la clave primaria*.
filename	El nombre del archivo que fue originalmente dado al archivo. Esto puede o no ser tomado en consideración, al determinar la ruta final del destino.

Tabla A3: *Argumentos opcionales de FileField*

storage: Un objeto para almacenamiento, el cual maneja el almacenamiento y recuperación de los archivos.

El widget predeterminado para este campo es un a ClearableFileInput.

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

1. En el archivo de configuración (settings.py), es necesario definir MEDIA_ROOT con la ruta completa al directorio donde quieras que Django almacene los archivos subidos. (Para mayor rendimiento, estos archivos no se almacenan en la base de datos.) Define MEDIA_URL con la URL pública base de ese directorio. Asegúrate de que la cuenta del usuario del servidor web tenga permiso de escritura en este directorio.

2. Agregar el FileField o ImageField al modelo, asegúrate de definir la opción upload_to para decirle a Django a cual subdirectorio de MEDIA_ROOT debe subir los archivos.
3. Todo lo que se va a almacenar en la base de datos es la ruta al archivo (relativa a MEDIA_ROOT). Seguramente preferirás usar la facilidad de la función url provista por Django. Por ejemplo, si tu campo "ImageField" se llama portada, puedes obtener la URL absoluta a la imagen en un plantilla usando:

```
{{ object.portada.url }}
```

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

Si quieres recuperar el nombre en disco del archivo subido, o una URL que se refiera a ese archivo, o el tamaño del archivo, puedes referirte al archivo mediante sus atributos name, url y size.

Cualquiera que sea la forma en que manejes los archivos subidos, tienes que prestar mucha atención a donde los estás subiendo y qué tipo de archivos son, para evitar huecos en la seguridad. *Valida todos los archivos subidos* para asegurarte que esos archivos son lo que piensas que son. Por ejemplo, si dejas que cualquiera suba archivos ciegamente, sin validación, a un directorio que está dentro de la raíz de documentos (*document root*) de tu servidor web, alguien podría subir un script CGI o PHP y ejecutarlo visitando su URL en tu sitio. ¡No permitas que pase!

Por defecto las instancias de FileField son creadas usando columnas varchar(100) en la base de datos. Como otros campos, puedes cambiar el máximo permitido, usando el argumento max_length.

FilePathField

Un campo cuyas opciones están limitadas a los nombres de archivo en un cierto directorio en el sistema de archivos. Tiene tres argumentos especiales, que se muestran en la Tabla A-4.

Argumento	Descripción
path	*Requerido*; la ruta absoluta en el sistema de archivos hacia el directorio del cual este FilePathField debe tomar sus opciones (ej.: "/home/images").
match	Opcional; una expresión regular como string, que FilePathField usará

	para filtrar los nombres de archivo. Observar que la regex será aplicada al nombre de archivo base, no a la ruta completa (ej.: "foo.*\.txt^", va a coincidir con un archivo llamado foo23.txt, pero no con bar.txt o foo23.gif).
recursive	Opcional; True o False. El valor por omisión es False. Especifica si deben incluirse todos los subdirectorios de path.
alow_files	Opcional, True o False. El valor por omisión es True Especifica si todos los los archivos de los directorios especificados deben ser incluidos. Utiliza este o allow_folders como True.
alow_folders	Opcional, True o False. El valor por omisión es False Especifica si todas las carpetas de los directorios especificados deben ser incluidas. Utiliza este o allow_files como True.

Tabla A4 *Tabla de opciones extra de FilePathField*

Por supuesto, estos argumentos pueden usarse juntos.

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

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

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

FloatField

Un número de punto flotante, representado en Python por una instancia de float.

IntegerField

Similar a FileField, pero valida que el objeto subido sea una imagen válida. Tiene dos argumentos opcionales extra:

1. **height_field**: Nombre del campo del modelo usado para auto-rellenar con la altura de la imagen cada vez que se guarde una instancia de una imagen.

2. **width_field**: Nombre del campo del modelo usado para auto-rellenar con ancho de la imagen cada vez que se guarde una instancia de una imagen.

Además de los atributos especiales requeridos que están disponibles para FileField, un ImageField puede contener atributos height y width que corresponden a la altura y al ancho de la imagen en pixeles.

Requiere PIL, la librería de Imágenes Python, en especial de Pillow.

Por defecto las instancias de ImageField son creadas usando columnas varchar(100) en la base de datos. Como con otros campos, puedes cambiar el máximo permitido, usando el argumento max_length.

IntegerField

Un entero. Valores desde -2147483648 hasta 2147483647 son soportados por Django de forma segura en todas las bases de datos.

GenericIPAddressField

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

NullBooleanField

Similar a BooleanField, pero permite None/NULL como opciones. Usar éste en lugar de un BooleanField con null=True.

PositiveIntegerField

Similar a IntegerField, pero debe ser positivo.

PositiveSmallIntegerField

Similar a PositiveIntegerField, pero solo permite valores por debajo de un límite. Valores desde 0 a 32767 son soportados de forma segura en Django.

SlugField

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

De igual forma que en CharField, puedes especificarlo con max_length. Si max_length no está especificado, Django asume un valor por omisión de 50.
Un SlugField implica db_index=True debido a que este tipo de campo se usa principalmente para búsquedas en la base de datos.

SlugField acepta una opción extra, prepopulate_from, que es una lista de campos a partir de los cuales auto-rellenar el slug, vía JavaScript, en el formulario de administración del objeto:

```
models.SlugField(prepopulate_from=("pre_name", "name"))
```

prepopulate_from no acepta nombres tipo DateTimeField como argumentos.

SmallIntegerField

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

TextField

Un campo de texto de longitud ilimitada.

TimeField

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

URLField

Un campo para una URL. Si la opción verify_exists es True (valor por omisión), se comprueba la existencia de la URL dada (la URL que arroja y no da una respuesta 404).

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

UUIDField

Un campo para almacenar Identificadores Universales Únicos. Usando la clase UUID de Python.

Los Identificadores Universales Únicos, son una buena alternativa para campos AutoField que usan una primary_key. La base de datos no genera el UUID, lo más recomendable es usarlos con un parámetro default:

```
import uuid
from django.db import models

class ModeloUUID(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    # Otros campos
```

Observa como el llamable (sin paréntesis) es pasado a default, no a una instancia de UUID.

Opciones Universales de Campo

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

null

Si está en True, Django almacenará valores vacíos como NULL en la base de datos. Si esta en False, los valores vacíos que se guarden resultaran probablemente en errores de la base de datos. El valor por omisión es False.

Observa que los valores de string nulo siempre se almacenan como strings vacíos, no como NULL. Utiliza null=True solo para campos no-string, como enteros, booleanos y fechas. En los dos casos, también necesitarás establecer blank=True si deseas permitir valores vacíos en los formularios, ya que el parámetro null solo afecta el almacenamiento en la base de datos (ver la siguiente sección, titulada “blank”).

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

■**Nota:** Cuando uses como base de datos un backend de Oracle, la opción null=True será usada para obligar a los campos basados en cadenas a aceptar cadena vacías como sea posible, y el valor NULL será almacenado para denotar la cadena vacía.

blank

Si está en True, está permitido que el campo esté en blanco. El valor por omisión es False.

Observa que esto es diferente de null. Null solo se relaciona con la base de datos, mientras que blank está relacionado con la validación. Si un campo tiene un campo blank=True, la validación del administrador de Django permitirá la entrada de un valor vacío. Si un campo tiene blank=False, es un campo requerido.

choices

Un iterable (ej.: una lista, tupla, o otro objeto iterable de Python) de dos tuplas para usar como opciones para este campo.

Si esto está dado, la interfaz de administración de Django utilizará un cuadro de selección en lugar del campo de texto estándar, y limitará las opciones a las dadas.

Una lista de opciones se ve así:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

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

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

```
class Foo(models.Model):
    GENDER_CHOICES = (
        ('M', 'Male'),
        ('F', 'Female'),
)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

O fuera de la clase del modelo:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)

class Foo(models.Model):
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

Para cada campo del modelo que tenga establecidas choices, Django agregará un método para recuperar el nombre legible por humanos para el valor actual del campo.

db_column

El nombre de la columna de la base de datos a usar para este campo. Si no está dada, Django utilizará el nombre del campo. Esto es útil cuando estás definiendo un modelo sobre una base de datos existente.

Si tu nombre de columna de la base de datos es una palabra reservada de SQL, o contiene caracteres que no están permitidos en un nombre de variable de Python (en particular el guión simple), no hay problema. Django quotea los nombres de columna y tabla detrás de la escena.

db_index

Si está en True, Django creará un índice en la base de datos para esta columna cuando cree la tabla (es decir, cuando ejecute manage.py migrate).

db_tablespace

El nombre de la tabla de la base de datos para usar en este índice de campo, si el campo tiene un índice.

default

El valor por omisión del campo.

editable

Si es False, el campo no será editable en la interfaz de administración o vía procesamiento de formularios. El valor por omisión es True.

error_messages

Argumento que permite sobrescribir el mensaje por omisión que el campo lanza.

help_text

Texto de ayuda extra a ser mostrado bajo el campo en el formulario de administración del objeto. Es útil como documentación aunque tu objeto no tenga formulario de administración.

primary_key

Si es True, este campo es la clave primaria del modelo.

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

```
d = models.AutoField('ID', primary_key=True)
```

Por lo tanto, no necesitas establecer primary_key=True en ningún campo, salvo que quieras sobreescribir el comportamiento por omisión de la clave primaria.

primary_key=True implica blank=False, null=False, y unique=True. Solo se permite una clave primaria en un objeto.

unique

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

unique_for_date

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

```
class Entrada(models.Model):
    fecha = models.DateTimeField()
    slug = models.SlugField(unique_for_date="fecha")
    #    ...
```

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

unique_for_month

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

unique_for_year

Similar a unique_for_date y unique_for_month, pero para el año.

verbose_name

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

En este ejemplo, el nombre descriptivo es " tu nombre ":

```
nombre =  models.CharField("tu nombre", max_length=30)
```

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

```
nombre = models.CharField(max_length=30)
```

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

```
class Encuesta(models.Model):

    pregunta = models.ForeignKey(Pregunta, verbose_name="preguntas relacionadas")
    sitio = models.ManyToManyField(Sitio, verbose_name="lista de sitios")
    lugar = models.OneToOneField(Lugar, verbose_name="lugares relacionados")
```

La convención es no capitalizar la primera letra del verbose_name. Django convertirá a mayúscula automáticamente la primera letra cuando lo necesite.

validators

Una lista de validadores que se ejecutaran para este campo.

Relaciones

Es claro que el poder de las bases de datos se basa en relacionar tablas entre sí. Django ofrece formas de definir los tres tipos de relaciones más comunes en las bases de datos: muchos-a-uno, muchos-a-muchos, y uno-a-uno.

Relaciones Muchos-a-Uno

Para definir una relación muchos-a-uno, usa un campo tipo ForeignKey. El cual se usa como cualquier otro tipo Field: incluyéndolo como un atributo de clase en tu modelo.
ForeignKey requiere un argumento posicional: la clase a la cual se relaciona el modelo.
Por ejemplo, si un modelo Carro tiene un Fabricante – es decir, un Fabricante fabrica múltiples carros pero cada Carro tiene solo un Fabricante – usa la siguiente definición:

```
class Fabricante(models.Model):
    #...

class Carro(models.Model):
    fabricante = models.ForeignKey(Fabricante)
    #...
```

Para crear una relación *recursiva* – un objeto que tiene una relación muchos-a-uno consigo mismo – usa models.ForeignKey('self'):

```
class Empleado(models.Model):
    manager = models.ForeignKey('self')
```

Si necesitas crear una relación con un modelo que aún no ha sido definido, puedes usar el nombre del modelo en lugar del objeto modelo:

```
from django.db import models

class Carro(models.Model):
    fabricante = models.ForeignKey('Fabricante')
    # ...

class Fabricante(models.Model):
    # ...
```

Para referirte a modelos que han sido definidos en otra aplicación, puedes explícitamente especificar un modelo con el nombre completo de la etiqueta de la aplicación. Por ejemplo si el modelo Manufactura arriba definido es definido en otra aplicación llamada producción, necesitas usar:

```
class Carro(models.Model):
   fabricante = models.ForeignKey('produccion.Fabricante')
```

Esta clase de referencia puede ser útil al resolver dependencias circulares de importaciones entre dos aplicaciones.

Toma en cuenta que cada vez que creas una relación foránea, se crea un un índice en la base de datos de forma automática. Puedes desactivar este comportamiento fijando db_index = False, en el archivo del modelo. Si quieres evitar la sobrecarga que ocasiona un índice, o si estas creando un índice alternativo.

☠**Advertencia:** No es recomendable tener un campo ForeignKey de una aplicación sin migraciones, enlazado a una aplicación con las migraciones aplicadas.

Detrás de escena, Django agrega "_id" al nombre de campo para crear su nombre de columna en la base de datos. En el ejemplo anterior, la tabla de la base de datos correspondiente al modelo Carro, tendrá una columna fabricante_id. (Puedes cambiar esto explícitamente especificando db_column; ver más arriba en la sección "db_column".) De todas formas, tu código nunca debe utilizar el nombre de la columna de la base de datos, salvo que escribas tus propias SQL. Siempre te manejarás con los nombres de campo de tu objeto modelo.

Se sugiere, pero no es requerido, que el nombre de un campo ForeignKey (fabricante en el ejemplo) sea el nombre del modelo en minúsculas. Por supuesto, puedes ponerle el nombre que quieras. Por ejemplo:

```
class Carro(models.Model):

   fabrica_de_autos = models.ForeignKey(Fabricante)
   # ...
```

Los campos ForeignKey reciben algunos argumentos extra para definir como debe trabajar la relación (ver Tabla A-5). Todos son opcionales.

Argumento	Descripción

limit_choices_to	Establece un límite a las opciones disponibles para el campo, cuando es renderizado usando ModelForm en el admin (por omisión, todos los objetos de el queryset están disponibles para elegir). Usa un diccionario, un objeto Q, o un llamable que devuelva un diccionario, o un objeto Q que puede ser utilizado.
	Por Ejemplo:
	
	`staff_member=models.ForeignKey(`
	`limit_choices_to={'is_staff': True}`
	
	Esto hace que el campo correspondiente en ModelForm liste únicamente Users que tengan asignado is_staff=True.

related_name	El nombre a utilizar para la relación desde el objeto relacionado hacia él objeto con el que se relaciona.

related_query_name	El nombre a utilizar para el nombre inverso del filtro del modelo. Remplaza el valor de related_name si se establece, de otra manera será el valor por defecto del nombre del modelo:
	
	`# Declara el campo foráneo con "related_query_name"`
	`class Tag(models.Model):`
	`articulo = models.ForeignKey(Articulo,`
	`related_name “tags”=related_query_name="tag")`
	`nombre = models.CharField(max_length=255)`
	`# Éste ahora es el nombre del filtro inverso`
	`Articulo.objects.filter(tag_name = "importante")`

to_field	El campo en el objeto relacionado con el cual se establece la relación. Por omisión, Django usa la clave primaria del objeto relacionado.
db_constraint	Regula la creación o no de restricciones en la base de datos para la clave foránea. Por omisión es True, la mayoría de las veces será cierto, Usar False puede ser muy malo para la integridad de los datos. Dicho esto, aquí hay algunos escenarios donde podrías querer hacer esto: Tienes datos heredados que no son validos. Tu base de datos está rota.

on_delete	Cuando un objeto referenciado por un campo ForeignKey es borrado. Django por omisión emula el comportamiento de las restricciones SQL ON DELETE CASCADE y borra también el objeto que contiene el campo ForeignKey.
	Este comportamiento puede ser sobrescrito usando algún argumento con on_delete como los siguientes:
	CASCADE: Borra en cascada, el comportamiento por omisión.
	PROTECT: Previene que se borre la referencia a un objeto.
	SET_NULL Fija un campo ForeignKey a null; siempre y cuando use null = True.
	SET_DEFAULT Fija un campo ForeignKey como el valor por omisión.
	SET() Fija un campo ForeignKey con el valor pasado a SET(), o mediante un llamable.
	Por ejemplo si quieres tener un campo ForeignKey que contenga valores nulos, cuando el objeto sea borrado utiliza lo siguiente:
	`user = models.ForeignKey(User, blank=True, null=True,`
	`on_delete=models.SET_NULL).`

DO_NOTHING	No toma ninguna acción.

Tabla A5 *Opciones de ForeignKey*

Relaciones Muchos a Muchos

Para definir una relación muchos-a-muchos, usa un campo ManyToManyField. Al igual que los campos ForeignKey, ManyToManyField requiere un argumento

posicional: la clase a la cual se relaciona el modelo, trabaja de igual forma que los campos ForeignKey, incluyendo relaciones recursivas y perezosas.

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

```
class Ingredientes(models.Model):
   #...

class Pizza(models.Model):
   ingredientes = models.ManyToManyField(Ingredientes)
   ...
```

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

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

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

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

Si estás usando la interfaz de administración, las instancias ManyToManyField deben ir en el objeto que va a ser editado en la interfaz de administración. En el ejemplo anterior, los ingredientes están en la Pizza (en lugar de que el Ingredientes tenga pizzas ManyToManyField) porque es más natural pensar que una Pizza tiene varios Ingredientes que pensar que un ingrediente está en muchas pizzas. En la forma en que está configurado el ejemplo, el formulario de administración de el objeto "Pizza" permitirá que los usuarios seleccionen los ingredientes.

Los objetos ManyToManyField toman algunos argumentos extra para definir como debe trabajar la relación (ver Tabla A-6). Todos son opcionales.

Argumento	Descripción
related_name	El nombre a utilizar para la relación desde el objeto relacionado hacia este objeto.
related_query_name	El nombre a utilizar para el nombre inverso del filtro del modelo. Remplaza el valor de related_name si se establece, de otra manera será el valor por defecto del nombre del modelo.
limit_choices_to	Ver la descripción en ForeignKey. limit_choices_to no tiene efecto cuando es usado con una tabla intermedia especificada usando el parámetro througth
symmetrical	Solo utilizado en la definición de ManyToManyField sobre sí mismo. Considera el siguiente modelo:

	class Persona(models.Model): friends = models.ManyToManyField("self") Cuando Django procesa este modelo, identifica que tiene un ManyToManyField sobre sí mismo, y como resultado, no agrega un atributo persona_set a la clase Persona. En lugar de eso, se asumen que el ManyToManyField es simétrico – esto es, si yo soy tu amigo, entonces tu eres mi amigo. Si no deseas la simetría en las relaciones ManyToMany con self, establece symmetrical en False. Esto forzará a Django a agregar el descriptor para la relación inversa, permitiendo que las relaciones ManyToMany sean asimétricas.
through	Django automáticamente genera una tabla para manejar las relaciones muchos a muchos. Si quieres especificar una tabla intermedia, puedes usar la opción through para especificar el modelo Django que representara la tabla intermedia que quieras usar. Los casos más comunes para usar esta opción es cuando quieres asociar datos extras con relaciones muchos a muchos. Si no especificas explícitamente un modelo con through, todavía hay implícita una clase del modelo through, que puedes usar directamente para crear las tablas y sus asociaciones. Tiene tres campos: id: La clave primaria de la relación. <containing_model>_id: El id del modelo que declara la relación ManyToManyField. <other_model>_id: El id del modelo que enlaza a ManyToManyField

through_fields	Usado únicamente en modelos intermedios personalizados. Django normalmente determina el campo intermedio de un modelo que se usara para establecer el orden en una relación muchos a muchos automáticamente. Considera el siguiente ejemplo:
	
	from django.db import models
	
	class Persona(models.Model):
	nombre = models.CharField(max_length=50)
	
	class Grupo(models.Model):
	nombre = models.CharField(max_length=128)
	miembros = models.ManyToManyField(Persona,
	through='mienbros', through_fields=('grupo',
	'persona'))
	
	class Mienbros(models.Model):
	grupo = models.ForeignKey(Group)
	persona = models.ForeignKey(Person)
	invitados = models.ForeignKey(Person,
	related_name="mienbros_invitados")
	razon_invitacion = models.CharField(max_length=64)
	
	Miembros tiene *dos* claves foráneas Persona (persona e

	invitados) lo cual hace que la relación sea ambigua y Django no sepa cual utilizar. En este caso, debes explícitamente especificar cual clave foránea deberá utilizar Django, usando through_fields, como en el modelo Grupo. through_fields acepta 2 tuplas ('field1', 'field2'), donde field1 es el nombre de la clave foránea del modelo de la clase ManyToManyField donde está definido con (grupo en este caso), y field2 es el nombre de la clave foránea con el campo del modelo. Cuando se usan más de una clave foránea en un modelo intermedio para cualquiera (o ambos) de los modelos participantes en la relación muchos a muchos, es necesario *especificar* through_fields. Esto también se aplica a relaciones recursivas, cuando un modelo intermedio es usado con más de dos claves foráneas en un modelo, o si deseas explícitamente especificar cuál de los dos debe usar Django.
swappable	Controla la reacción del framework de migraciones, si el campo ManyToManyField apunta a un modelo intercambiable o swappable, Si este es True –el valor por omisión, el campo *ManyToManyField*` apunta al modelo con el cual coincide el valor actual de settings. AUTH_USER_MODEL (u otra configuración de un modelo de intercambio), la relación será almacenada en la migración usando una referencia a la configuración, no al modelo directamente. Únicamente querrías sobrescribir esto a False si estás seguro de que tu modelo apunta siempre hacia el modelo intercambiable – por ejemplo, si es un modelo diseñado específicamente para un modelo de usuario personalizado. Por último, toma en cuenta que los campos ManyToManyField no soportan validadores y que null no tiene efecto ya que no es requerido en una relación a nivel de base de datos.
db_table	El nombre de la tabla a crear para almacenar los datos de la relación muchos-a-muchos. Si no se provee, Django asumirá un nombre por omisión basado en los nombres de las dos tablas a ser vinculadas.
db_constraint	Como en los campos ForeignKey, regula la creación o no de restricciones en la base de datos para la clave foránea. El valor por defecto es True

Tabla A6 Opciones de un campo ManyToManyField

Relaciones uno a uno

Una relación uno-a-uno, es conceptualmente muy parecida a una relación foránea o ForeignKey con un parámetro unique=True, solo que el lado "inverso" de la relación devuelve directamente un único objeto.

Esto es más útil como la clave primaria de un modelo que "extiende" otro modelo de la misma forma; la herencia multi-tablas es implementada agregando implícitamente una relación uno a uno del modelo hijo al modelo padre.

Un argumento posicional es requerido, la clase a la cual el modelo se relaciona. Esto funciona exactamente de la misma forma en que lo hace para ForeignKey incluyendo todas las opciones incluyendo relaciones recursivas y perezosas.

Adicionalmente OneToOneField acepta todos los argumentos extras aceptados por un campo ForeignKey, más un argumento extra.

parent_link: Cuando es True y es usado en un modelo el cual hereda de otro modelo (concretamente) indica que el campo se debe utilizar para enlazar la clase padre, en lugar de el extra OneToOneField que normalmente sería creado implícitamente por la subclase

Si no especificas el nombre para el argumento related_name en un campo OneToOneField, Django usara el nombre del modelo en minúsculas, como el valor por default. Por ejemplo:

```
from django.conf import settings
from django.db import models

class UsuarioEspecial(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    supervisor = models.OneToOneField(settings.AUTH_USER_MODEL,
        related_name='supervisor')
```

El modelo User tendrá los siguientes atributos:

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'usuarioespecial')
True
>>> hasattr(user, 'supervisor')
True
```

Una excepción del tipo DoesNotExist es lanzada cuando se accede a la relación inversa de una entrada de una tabla que no existe. Por ejemplo, si un user no tiene un supervisor definido en UsuarioEspecial:

```
>>> user.supervisor
Traceback (most recent call last):
...
DoesNotExist: User matching query does not exist.
```

Opciones de los Metadatos de un Modelo

Los metadatos específicos de un modelo viven en una clase *Meta* definida en el cuerpo de la clase del modelo:

```
class Libro(models.Model):
    titulo = models.CharField(max_length=100)

    class Meta:
    # Los metadatos van aquí
    ...
```

Los metadatos del modelo son "*cualquier cosa que no sea un campo*", como opciones de ordenamiento, nombre en plural, etc.

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

abstract

Si es True, este modelo será una clase base abstracta.

db_table

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

Para ahorrarte tiempo, Django deriva automáticamente el nombre de la tabla de la base de datos a partir del nombre de la clase modelo y la aplicación que la contiene. Un nombre de tabla de base de datos de un modelo se construye uniendo la etiqueta de la aplicación del modelo – el nombre que usaste en manage.py startapp – con el nombre de la clase modelo, con un guión bajo entre ellos.

Por ejemplo, si tienes una aplicación biblioteca (creada por manage.py startapp biblioteca), un modelo definido como class Libro tendrá una tabla en la base de datos llamada libros.

Para sobreescribir el nombre de la tabla de la base de datos, use el parámetro db_table dentro de class Meta:

```
class Libro(models.Model):
...
    class Meta:
        db_table = 'un_nombre_cualquiera'
```

Si no se define, Django utilizará app_label + '_' + model_class_name.

Si tu nombre de tabla de base de datos es una palabra reservada de SQL, o contiene caracteres que no están permitidos en los nombres de variable de Python (especialmente el guión simple), no hay problema. Django entrecomilla los nombres de tabla y de columna detrás de la escena.

db_tablespace

El nombre de la tabla de la base de datos para usar por el modelo. Si el backend no soporta tablespaces, esta opción será ignorada.

get_latest_by

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

Aquí hay un ejemplo:

```
get_latest_by = "fecha_publicacion"
```

managed

Por defecto es True, lo cual significa que Django creará las apropiadas tablas de la base de datos en las migraciones, o como parte de las migraciones y las removerá como parte del comando flush. Es decir, Django *maneja* los ciclos vitales de las tablas de la base de datos.

Si es False, no se realizará ningunas operaciones de creación o borrado de la tabla de base de datos para este modelo. Esto es útil si el modelo representa una tabla existente o una vista de la base de datos que se ha creado por algún otro medio. Ésta es la única diferencia cuando usas managed=False. El resto de los aspectos del manejo del modelo son exactamente iguales que los de uno normal. Esto incluye

Agregar una clave primaria automáticamente al modelo si no se declara una. Para evitar confusiones, para los que lean el código, es recomendable especificar todas las columnas de la base de datos que se están modelando, aun cuando se estén usando modelos unmanaged

Si el modelo con managed=False contiene un campo ManyToManyField que enlace otro modelo unmanaged, en lugar de una tabla intermedia para juntar la tabla muchos a muchos esta no se creara. Sin embargo, una tabla intermediaria entre un modelo managed y un unmanaged si se puede crear.

Si necesitas cambiar este comportamiento por defecto, crea una tabla intermedia como un modelo explicito (con managed según lo necesitado) y usa el atributo through para hacer que la relación use un modelo personalizado.

Para pruebas que incluyan modelos con managed=False, necesitas asegurarte que se están creando las tablas correctas, como parte de las pruebas.

Si estas interesado en cambiar el comportamiento a nivel-Python de un modelo de una clase, *puedes* usar managed=False y crear una copia de un modelo existente. Sin embargo existe una mejor forma de aprovechar esta situación: usa proxy-models

ordering

El ordenamiento por omisión del objeto, utilizado cuando se obtienen listas de objetos:
ordering = ['-fecha_publicacion']

Esta es una tupla o lista de cadenas o strings. Cada string es un nombre de campo con un prefijo opcional -, que indica orden descendiente. Los campos sin un - precedente se ordenarán en forma ascendente. Use el string "?" para ordenar al azar.

Nota: No es recomendable tener un campo ForeignKey de una aplicación sin migraciones, enlazado a una aplicación con las migraciones aplicadas.

Sin importar de cuántos campos consista *ordering*, el sitio administrativo únicamente usara el primer campo.

Por ejemplo, para ordenar por un campo titulo en orden ascendente (A-Z), usa esto:

```
ordering = ['titulo']
```

Para ordenar por título en orden descendente (Z-A), usa esto:

```
ordering = ['-titulo']
```

Para ordenar por título en orden descendente, y luego por autor en orden ascendente, usa esto:

```
ordering = ['-titulo', 'autor']
```

order_with_respect_to

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

```
Class  Respuesta(models.Model):
   pregunta = models.ForeignKey(Pregunta)
   # ...

   class Meta:
   order_with_respect_to = 'pregunta'
```

*order_with_respect_*to agrega un campo a la base de datos, en específico una columna llamada _order, asi que asegúrate que los cambios sea aplicados correctamente en cada migración, si agregas o cambias order_with_respect_to después de la inicial migración.

proxy

Si se establece en True, un modelo de una subclase de otro modelo, será tratado como un modelo proxi.

unique_together

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

```
unique_together = (("driver", "restaurante"),)
```

Esto es una lista de listas de campos que deben ser únicos cuando se consideran juntos. Es usado en la interfaz de administración de Django y se refuerza a nivel de base de datos (esto es, se incluyen las sentencias UNIQUE apropiadas en la sentencia CREATE TABLE).

verbose_name

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

```
verbose_name = "pizza"
```

Si no se define, Django utilizará una versión adaptada del nombre de la clase, en la cual CamelCase se convierte en camel case.

verbose_name_plural

El nombre del objeto en plural:

```
verbose_name_plural = "historias"
```

Si no se define, Django agregará una "*s*" al final del verbose_name.

Opciones del Administrador

ModelAdmin

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

Usualmente se almacena en el archivo *admin.py* dentro de la aplicación (el comando startapp crea el archivo admin.py).

Por ejemplo:

```
from biblioteca.models import Autor

class InterfazAutor(admin.ModelAdmin):
   pass

admin.site.register(Author, InterfazAutor)
```

De forma predeterminada la clase ModelAdmin registra todos los valores del modelo, si esto es lo que buscas una manera muy sencilla de simplificar el ejemplo anterior es usando lo siguiente

```
from django.contrib import admin
from bibloiteca.models import Autor

 admin.site.register(Autor)
```

También puedes usar un decorador, para registrar la clase ModelAdmin, directamente en el modelo:

```
from django.contrib import admin
from .models import Autor

@admin.register(Author)
class AuthorAdmin(admin.ModelAdmin):
   pass
```

Las siguientes secciones presentan una lista de alguna de las opciones que acepta la clase ModelAdmin para personalizar sus atributos. Ninguna de estas opciones es requerida. Para utilizar una interfaz de administración.

date_hierarchy

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

Aquí hay un ejemplo:

```
# Archivo models.py
class Libro(models.Model):
    fecha = models.DateTimeField()
    ...


# Archivo admin.py
from biblioteca.models import Autor

class InterfazLibro(admin.ModelAdmin):
    date_hierarchy = "fecha"

admin.site.register(Libro, InterfazLibro)
```

Este atributo, establece la lista de nombres de campos que se deben excluir de un formulario.

Por ejemplo, considera el siguiente modelo:

```
from django.db import models

class Autor(models.Model):
    nombre = models.CharField(max_length=100)
    titulo = models.CharField(max_length=3)
    fecha_nacimiento = models.DateField(blank=True, null=True)
```

Si solo quieres incluir en un formulario del modelo Autor los campos nombre y titulo, puedes especificarlos a través de exclude de la siguiente forma:

```
from django.contrib import admin

class AutorAdmin(admin.ModelAdmin):
    fields = ('nombre', 'titulo')


class AutorAdmin(admin.ModelAdmin):
   exclude = ('fecha_nacimiento',)
```

fields

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

Para mostrar múltiples campos en la misma linea, encierra esos campos en su propia tupla. En este ejemplo, los campos nombre y apellido se mostrarán en la misma línea.

Por ejemplo, para definir un simple formulario de un modelo de django.contrib.flatpages.models.FlatPage, podemos usar fields así:

```
class FlatPageAdmin(admin.ModelAdmin):
    fields = ('url', 'title', 'content')
```

Nota: La opcion fields no deve de confundirse con el diccionario fields de el atributo de la opcion fieldsets, de la siguiente seccion.

fieldsets

Establece fieldsets para controlar la disposición de las páginas "agregar" y "modificar" de la interfaz de administración.

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

```
from django.contrib import admin

class FlatPageAdmin(admin.ModelAdmin):
   fieldsets = (
      (None, {
         'fields': ('url', 'title', 'content', 'sites')
   }),
   ('Advanced options', {
       'classes': ('collapse',),
        'fields': ('enable_comments', 'registration_required', 'template_name')
   }),
)
```

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

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

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

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

- **fields**

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

  ```
  {
  'fields': ('nombre', 'apellido', 'domicilio', 'ciudad', 'estado'),
  }
  ```

  Para mostrar múltiples campos en la misma linea, encierra esos campos en su propia tupla.:

  ```
  {
  'fields': (('nombre', 'apellido'), 'domicilio', 'ciudad', 'estado'),
  }
  ```

- **clases**

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

Ejemplo:

```
{
'classes': ('wide', 'extrapretty'),
}
```

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

- **description**

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

list_display

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

Ejemplo:

```
list_display = ('nombre', 'apellido')
```

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

Aquí hay algunos casos especiales a obsevar acerca de list_display:

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

Aquí está un modelo de ejemplo completo:

```
from django.db import models
from django.contrib import admin
from django.utils.html import format_html

class Persona(models.Model):
    nombre = models.CharField(max_length=50)
```

```
    apellido = models.CharField(max_length=50)
    color_codigo = models.CharField(max_length=6)

    def nombre_coloreado(self):
        return format_html('<span style="color: #{0};">{1} {2}</span>',
                    self.color_codigo,
                    self.nombre,
                    self.apellido)

    nombre_coloreado.allow_tags = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'apellido', 'nombre_coloreado')
```

Si el string dado es un método del modelo que retorna True o False, Django mostrará un bonito icono "on" o "off" si le das al método un atributo boolean con valor en True.

Aquí está un modelo de ejemplo completo:

```
class Person(models.Model):
    nombre = models.CharField(maxlength=50)
    fecha_nacimiento = models.DateField()

class Admin:
    list_display = ('nombre', 'fecha_nacimiento')

    def nacido_cincuentas(self):
    return self.fecha_nacimiento.strftime('%Y')[:3] == 5

    nacido_cincuentas.boolean = True
```

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

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

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

Django trata de interpretar cada elemento *de list_display* en este orden:

- Un campo de un modelo.
- Un llamable.
- Una cadena de representación de un atributo ModelAdmin.
- Una cadena de representación de un atributo de un modelo.

list_display_links

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

Por omisión, la página de la lista de cambios vinculará la primera columna – el primer campo especificado en list_display – a la página de cambios de cada ítem.

Pero list_display_links te permite cambiar cuáles columnas se vinculan. Establece list_display_links a una lista o tupla de nombres de campo (en el mismo formato que list_display) para vincularlos.
list_display_links puede especificar uno o varios nombres de campo. Mientras los nombres de campo aparezcan en list_display, a Django no le preocupa si los campos vinculados son muchos o pocos. El único requerimiento es que si quieres usar list_display_links, debes definir list_display.

En este ejemplo, los campos nombre y apellido serán vinculados a la página de la lista de cambios:

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

    class Admin:
        list_display = ('nombre', 'apellido', 'fecha_cumpleaños')
        list_display_links = ('nombre', 'apellido')
```

En este ejemplo, la lista de cambios no tiene links:

```
class PersonaAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'apellido')
    list_display_links = None
```

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

list_filter

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

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

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

    class Admin:
        list_display = ('username', 'email', 'nombre', 'apellido', 'is_staff')
        list_filter = ('is_staff', 'is_superuser')

    ModelAdmin.list_per_page
```

list_per_page

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

list_select_related

Establece list_select_related para indicarle a Django que use select_related() al recuperar la lista de objetos de la página de la lista de cambios del administrador.

Esto puede ahorrarte una cantidad de consultas a la base de datos si estás utilizando objetos relacionados en la lista de cambios que muestra el administrador.

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

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

ordering

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

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

save_as

Establece save_as a True para habilitar la característica "save as" en los formularios de cambios del administrador.

Normalmente, los objetos tienen tres opciones al guardar: "Save", "Save and continue editing" y "Save and add another". Si save_as es True, "Save and add another" será reemplazado por un botón "Save as".

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

Por omisión, save_as es False.

save_on_top

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

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

Por omisión, save_on_top es False.

search_fields

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

Estos campos deben ser de alguna tipo de campo de texto, como CharField o TextField.

También puedes realizar una búsqueda relacionada sobre una relación ForeignKey con la notación de búsqueda de la API:

```
class Empleado(models.Model):
    departmento = models.ForeignKey(Departmento)
    ...

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

Cuando alguien hace una búsqueda en el cuadro de búsqueda del administrador, Django divide la consulta de búsqueda en palabras y retorna todos los objetos que contengan alguna de las palabras, sin distinguir mayúsculas y minúsculas, donde

cada palabra debe estar en al menos uno de los search_fields. Por ejemplo, si search_fields es ['nombre', 'apellido'] y un usuario busca john lennon, Django hará el equivalente a esta cláusula WHERE en SQL:

```
WHERE (nombre ILIKE '%john%' OR apellido ILIKE '%john%')
AND (nombre ILIKE '%lennon%' OR apellido ILIKE '%lennon%')
```

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

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

Tabla A7 *Operadores Permitidos en search_fields*

APÉNDICE B

Referencia de la API de base de datos

La API de base de datos de Django es la otra mitad de la API de modelos discutido en el *apéndice A*: Una vez que hayas definido un modelo, usarás esta API en cualquier momento que necesites acceder a la base de datos. A lo largo de todos estos capítulos, has visto ejemplos del uso de esta API; este apéndice explica las distintas opciones detalladamente.

De manera similar a lo que ocurre con las APIs de los modelos descritos en el apéndice anterior, estas APIs son considerados muy estables, aunque los desarrolladores de Django constantemente añaden nuevos atajos y conveniencias. Es buena idea consultar siempre la documentación en línea más actual que está disponible en: http://www.djangoproject.com/documentation/.

A lo largo de este apéndice, vamos a hacer referencia a los siguientes modelos, los cuales pueden formar una simple aplicación de un blog:

blog/models.py

```
from django.db import models

class Blog(models.Model):
    nombre = models.CharField(max_length=100)
    etiqueta = models.TextField()

    def __str__(self):              # __unicode__ on Python 2
        return self.nombre

class Autor(models.Model):
   nombre = models.CharField(max_length=50)
   email = models.EmailField(blank=True)

   def __str__(self):               # __unicode__ on Python 2
        return self.nombre

class Entrada(models.Model):
    blog = models.ForeignKey(Blog)
    titulo = models.CharField(max_length=255)
    texto = models.TextField()
    fecha_publicacion = models.DateField()
    autores = models.ManyToManyField(Autor)
    n_comentarios = models.IntegerField(blank=True, null=True)
    rating = models.IntegerField(blank=True, null=True)

    def __str__(self): # __unicode__ on Python 2
        return self.titulo
```

Creando Objetos

Para crear un objeto, crea una instancia de la clase modelo usando argumentos de palabra clave y luego llama a save() para grabarlo en la base de datos:

```
>>> from blog.models import Blog
>>> b = Blog(nombre='Beatles Blog', etiqueta='Las ultimos novedades de los beatles.')
>>> b.save()
```

Esto, detrás de escena, ejecuta una sentencia SQL INSERT. Django no accede a la base de datos hasta que tú explícitamente invoques a save().

El método save() no retorna nada.

Para crear un objeto y grabarlo todo en un paso revisa el método create de la clase Manager que describiremos en breve.

¿Qué pasa cuando grabas un objeto?

Cuando grabas un objeto, Django realiza los siguientes pasos:

1. **Emite una señal pre_save:** Se envía una notificación django.db.models.signals.pre_save informando que un objeto está a punto de ser grabado, permitiéndole a cualquier función estar atenta ante esta señal para tomar ciertas acciones para requisitos particulares.

2. **Pre-procesar los datos:** Se le solicita a cada campo del objeto implementar cualquier modificación automatizada de datos que pudiera necesitar realizar.

   La mayoría de los campos no realizan pre-procesamiento – los datos del campo se guardan tal como están. Sólo se usa pre-procesamiento en campos que tienen un comportamiento especial, como campos de archivo.

3. **Preparar los datos para la base de datos:** Se le solicita a cada campo que provea su valor actual en un tipo de dato que puede ser grabado en la base de datos.

   La mayoría de los campos no requieren preparación de los datos. Los tipos de datos simples, como enteros y cadenas, están "listos para escribir" como un objeto de Python. Sin embargo, tipo de datos más complejos requieren a menudo alguna modificación.

   Por ejemplo la clase *DateField* usa un objeto datetime Python para almacenar datos. Las bases de datos no almacenan objetos datetime, de manera que el valor del campo debe ser convertido en una cadena de fecha que cumpla con la norma ISO correspondiente para la inserción en la base de datos.

4. **Insertar los datos en la base:** Los datos pre-procesados y preparados son entonces incorporados en una sentencia SQL para su inserción en la base de datos.

5. **Emitir una señal post_save:** Como con la señal pre_save, esta es utilizada para proporcionar notificación de que un objeto ha sido grabado satisfactoriamente.

Claves primarias autoincreméntales

Por conveniencia, a cada modelo se le da una clave primaria autoincremental llamada id a menos que explícitamente especifiques primary_key=True en el campo (ver la sección titulada "*AutoField*" en el Apéndice B).

Si tu modelo tiene un AutoField, ese valor incrementado automáticamente será calculado y grabado como un atributo de tu objeto la primera vez que llames a save():

```
>>> b2 = Blog(nombre='Charlas Cheddar', etiqueta='Pensamientos sobre quesos.')
>>> b2.id # Devuelve "None", porque b no tiene un ID.
None
>>> b2.save()
>>> b2.id # Devuelve el ID de el nuevo objeto.
2
```

No hay forma de saber cuál será el valor de un identificador (ID) antes que llames a el metodo save() esto se debe a que ese valor es calculado por la base de datos, no por Django.

Si un modelo tiene un AutoField pero quieres definir el identificador de un nuevo objeto explícitamente cuando grabas, solo defínelo explícitamente antes de grabarlo en vez de confiar en la asignación automática de valor del identificador:

```
>>> b3 = Blog(id=3, nombre='Charlas Cheddar', etiqueta='Pensamientos sobre
quesos.')
>>> b3.id
3
>>> b3.save()
>>> b3.id
3
```

Si asignas manualmente valores de claves primarias autoincrementales ¡Asegúrate de no usar un valor de clave primaria que ya existe! Si creas un objeto con un valor explícito de clave primaria que ya existe en la base de datos, Django asumirá que estás cambiando el registro existente en vez de crear uno nuevo.

Dado el ejemplo precedente de blog 'Charlas Cheddar', este ejemplo sobrescribiría el registro previo en la base de datos:

```
>>> b4 = Blog(id=3, nombre='blog ', etiqueta='Todo menos quesos.')
>>> b4.save() # ¡Sobrescribe el anterior blog con ID=3!
```

El especificar explícitamente valores de claves primarias autoincrementales es más útil cuando se están grabando objetos en lotes, cuando estás seguro de que no tendrás colisiones de claves primarias.

Grabando cambios de objetos

Para grabar los cambios hechos a un objeto que existe en la base de datos, usa save().

Dada la instancia de Blog b4 que ya ha sido grabada en la base de datos, este ejemplo cambia su nombre y actualiza su registro en la base:

```
>>> b4.nombre = 'El mejor Blog'
>>> b4.save()
>>> b4
```

```
<Blog: El mejor Blog>
```

Detrás de escena, esto ejecuta una sentencia SQL UPDATE. De nuevo: Django no accede a la base de datos hasta que llamas explícitamente a save().

¿Cómo sabe django cuando usar UPDATE y cuando usar INSERT?

Habrás notado que los objetos de base de datos de Django usan el mismo método save() para crear y cambiar objetos. Django abstrae la necesidad de usar sentencias SQL INSERT o UPDATE. Específicamente, cuando llamas a save(), Django sigue este algoritmo:

- Si el atributo clave primaria del objeto tiene asignado un valor que evalúa True (esto es, un valor distinto a None o a la cadena vacía) Django ejecuta una consulta SELECT para determinar si existe un registro con la clave primaria especificada.
- Si el registro con la clave primaria especificada ya existe, Django ejecuta una consulta UPDATE.
- Si el atributo clave primaria del objeto *no* tiene valor o si lo tiene pero no existe un registro, Django ejecuta un INSERT.

Debido a esto, debes tener cuidado de no especificar un valor explícito para una clave primaria cuando grabas nuevos objetos si es que no puedes garantizar que el valor de clave primaria está disponible para ser usado.

La actualización de campos ForeignKey funciona exactamente de la misma forma; simplemente asigna un objeto del tipo correcto al campo en cuestión:

```
>>> from blog.models import Autor, Entrada
>>> joe = Autor.objects.create(nombre="Joe")
>>> Entrada.autor = joe
>>> joe.save()
```

Django se quejará si intentas asignar un objeto del tipo incorrecto.

Recuperando objetos

A través del libro has visto cómo se recuperan objetos usando código como el siguiente:

```
>>> from blog.models import Autor
>>> blogs = Autor.objects.filter(nombre__contains="Joe")
```

Hay bastantes partes móviles detrás de escena aquí: cuando recuperas objetos de la base de datos, estás construyendo realmente un QuerySet (consulta) usando el Manager del modelo. Este QuerySet sabe como ejecutar SQL y retornar los objetos solicitados.

El Apéndice A trató ambos objetos desde el punto de vista de la definición del modelo; ahora vamos a ver cómo funcionan.

Un QuerySet representa una colección de objetos de tu base de datos. Puede tener cero, uno, o muchos filtros – criterios que limitan la colección basados en parámetros

provistos. En términos de SQL un QuerySet se compara a una declaración SELECT y un filtro es una cláusula de limitación como por ejemplo WHERE o LIMIT.

Consigues un QuerySet usando el Manager del modelo. Cada modelo tiene por lo menos un Manager y tiene, por omisión, el nombre objects. Accede al mismo directamente a través de la clase del modelo, así:

```
>>> Blog.objects
<django.db.models.manager.Manager object at 0x137d00d>
```

Los Managers solo son accesibles a través de las clases de los modelos, en vez desde una instancia de un modelo, para así hacer cumplir con la separación entre las operaciones a "nivel de tabla" y las operaciones a "nivel de registro":

```
>>> b = Blog(nombre='Foo', etiqueta='Bar')
>>> b.objects
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: "Manager isn't accessible via % instances" " type...
```

El Manager es la principal fuente de QuerySets para un modelo. Actúa como un QuerySet "raíz" que describe todos los objetos de la tabla de base de datos del modelo. Por ejemplo, Blog.objects es el QuerySets inicial que contiene todos los objetos Blog en la base de datos.

Cacheo y QuerySets

Cada QuerySet contiene un cache, para minimizar el acceso a la base de datos. Es importante entender cómo funciona, para escribir código más eficiente.

En un QuerySet recién creado, el cache está vacío. La primera vez que un QuerySet es evaluado – y, por lo tanto, ocurre un acceso a la base de datos – Django graba el resultado de la consulta en el cache del QuerySet y retorna los resultados que han sido solicitados explícitamente (por ejemplo, el siguiente elemento, si se está iterando sobre el QuerySet).

Evaluaciones subsecuentes del QuerySet re-usan los resultados alojados en el cache.

Ten presente este comportamiento de caching, porque puede morderte si no usas tus QuerySets correctamente. Por ejemplo, lo siguiente creará dos QuerySets, los evaluará, y los descartará:

```
>>> print ([e.titulo for e in Entrada.objects.all()])
>>> print ([e.fecha_publicacion for e in Entrada.objects.all()])
```

Eso significa que la consulta sera ejecutada dos veces en la base de datos, duplicando la carga sobre la misma. También existe una posibilidad de que las dos listas pudieran no incluir los mismos registros de la base de datos, porque se podría haber agregado o borrado un Entrada durante el pequeñísimo período de tiempo entre ambas peticiones.

Para evitar este problema, simplemente graba el QuerySet y re-úsalo:

```
>>> from blog.models import Autor
>>> queryset = Autor.objects.all()
>>> print ([p.nombre for p in queryset])   # Evalua el query set.
>>> print ([p.email for p in queryset])   # Reusa la cache de la evaluación anterior.
```

Filtrando objetos

La manera más simple de recuperar objetos de una tabla es conseguirlos todos. Para hacer esto, usa el método all() en un Manager:

```
>>> from blog.models import Entrada
>>> Entrada.objects.all()
```

El método all() retorna un QuerySet de todos los objetos de la base de datos.

Sin embargo, usualmente solo necesitarás seleccionar un subconjunto del conjunto completo de objetos. Para crear tal subconjunto, refinas el QuerySet inicial, añadiendo condiciones con filtros. Usualmente harás esto usando los métodos filter() y/o exclude():

```
>>> si2014 = Entrada.objects.filter(fecha_publicacion__year=2014)
>>> no2014 = Entrada.objects.exclude(fecha_publicacion__year=2014)
```

Tanto filter() como exclude() toman argumentos de *patrones de búsqueda*, los cuales se discutirán detalladamente en breve.

Encadenando filtros

El resultado de refinar un QuerySet es otro QuerySet así que es posible enlazar refinamientos, por ejemplo:

```
>>> import datetime
>>> qs = Entrada.objects.filter(titulo__startswith='Que')
>>> qs = qs.exclude(fecha_publicacion__gte=datetime.datetime.now())
>>> qs = qs.filter(fecha_publicacion__gte=datetime.datetime(2005, 1, 1))
```

Esto toma el QuerySet inicial de todas las entradas en la base de datos, agrega un filtro, luego una exclusión, y luego otro filtro. El resultado final es un QuerySet conteniendo todas las entradas con un título que empieza con "What" que fueron publicadas entre Enero 1, 2005, y el día actual.

Es importante precisar aquí que los QuerySet son perezosos – el acto de crear un QuerySet no implica ninguna actividad en la base de datos. De hecho, las tres líneas precedentes no hacen *ninguna* llamada a la base de datos; puedes enlazar/encadenar filtros todo el día y Django no ejecutará realmente la consulta hasta que el QuerySet sea *evaluado*.

Puedes evaluar un QuerySet en cualquiera de las siguientes formas:

- **Iterando**: Un QuerySet es iterable, y ejecuta su consulta en la base de datos la primera vez que iteras sobre él. Por ejemplo, el siguiente QuerySet no es evaluado hasta que se iterado sobre él en el bucle for:

  ```
  >>> qs = Entrada.objects.filter(fecha_publicacion__year=2006)
  >>> qs = qs.filter(titulo__icontains="bill")
  >>> for e in qs:
  >>>     print (e.titulo)
  ```

  Esto imprime todos los títulos desde el 2006 que contienen "bill" pero genera solo un acceso a la base de datos.

- **Imprimiéndo:** Un QuerySet es evaluado cuando ejecutas repr() sobre el mismo. Esto es por conveniencia en el interprete interactivo Python, así puedes ver inmediatamente tus resultados cuando usas el API interactivamente.

- **Rebanado**: Según lo explicado en la próxima sección "Limitando QuerySets", un QuerySet puede ser rebanado usando la sintaxis de rebanado de arreglos de Python. Usualmente el rebanar un QuerySet retorna otro QuerySet (no evaluado), pero Django ejecutará la consulta a la base de datos si usas el parámetro "step" de la sintaxis de rebanado.

- **Convirtiendo a una lista**: Puedes forzar la evaluación de un QuerySet ejecutando list() sobre el mismo, por ejemplo:

```
>>> entrada_list = list(Entrada.objects.all())
```

Sin embargo, quedas advertido de que esto podría significar un gran impacto en la memoria porque Django cargará cada elemento de la lista en memoria. En cambio, el iterar sobre un QuerySet sacará ventaja de tu base de datos para cargar datos e inicializar objetos solo a medida que vas necesitando los mismos.

LOS QUERYSETS FILTRADOS SON ÚNICOS

Cada vez que refinas un QuerySet obtienes un nuevo QuerySet que no está de ninguna manera atado al *QuerySet`* anterior. Cada refinamiento crea un QuerySet separado y distinto que puede ser almacenado, usado y re-usado:

```
>>> q1 = Entrada.objects.filter(titulo__startswith="Que")
>>> q2 = q1.exclude(fecha_publicacion__gte=datetime.now())
>>> q3 = q1.filter(fecha_publicacion__gte=datetime.now())
```

Estos tres QuerySets son separados. El primero es un QuerySet base que contiene todas las entradas que contienen un título que empieza con "What". El segundo es un sub-conjunto del primero, con un criterio adicional que excluye los registros cuyo fecha_publicacion es mayor que el día de hoy. El tercero es un sub-conjunto del primero, con un criterio adicional que selecciona solo los registros cuyo fecha_publicacion es mayor que el día de hoy. El QuerySet inicial (q1) no es afectado por el proceso de refinamiento.

Limitando QuerySets

Usa la sintaxis de rebanado de arreglos de Python para limitar tu QuerySet a un cierto número de resultados. Esto es equivalente a las clausulas de SQL de LIMIT y OFFSET.

Por ejemplo, esto retorna las primeras cinco entradas (LIMIT 5):

```
>>> Entrada.objects.all()[:5]
```

Esto retorna las entradas desde la sexta hasta la décima (OFFSET 5 LIMIT 5):

```
>>> Entrada.objects.all()[5:10]
```

Generalmente, el rebanar un QuerySet retorna un nuevo QuerySet – no evalúa la consulta. Una excepción es si usas el parámetro "step" de la sintaxis de rebanado de Python. Por ejemplo, esto realmente ejecutaría la consulta con el objetivo de retornar una lista, objeto de por medio de los primeros diez:

```
>>> Entrada.objects.all()[:10:2]
```

Para recuperar *un* solo objeto en vez de una lista (por ej. SELECT foo FROM bar LIMIT 1) usa un simple índice en vez de un rebanado. Por ejemplo, esto retorna el primer Entrada en la base de datos, después de ordenar las entradas alfabéticamente por título:

```
>>> Entrada.objects.order_by('titulo')[0]
```

y es equivalente a lo siguiente:

```
>>> Entrada.objects.order_by('titulo')[0:1].get()
```

Nota, sin embargo, que el primero de estos generará IndexError mientras el segundo generará DoesNotExist si ninguno de los objetos coincide con el criterio dado.

Métodos de consulta que retornan nuevos QuerySets

Django provee una variedad de métodos de refinamiento de QuerySet que modifican ya sea los tipos de resultados retornados por el QuerySet o la forma como se ejecuta su consulta SQL.

Estos métodos se describen en las secciones que siguen. Algunos de estos métodos reciben argumentos de patrones de búsqueda, los cuales se discuten en detalle más adelante.

filter(**lookup)

Retorna un nuevo QuerySet conteniendo objetos que son iguales a los parámetros de búsqueda provistos.

exclude(**kwargs)

Retorna un nuevo QuerySet conteniendo objetos que *no* son iguales a los parámetros de búsqueda provistos.

order_by(*fields)

Por omisión, los resultados retornados por un QuerySet están ordenados por la tupla de ordenamiento indicada por la opción ordering en los metadatos del modelo.

Puedes sobrescribir esto para una consulta particular usando el método order_by():

```
>>>Entrada.objects.filter(fecha_publicacion__year=2005).order_by(
        '-fecha_publicacion', 'titulo')
```

Este resultado será ordenado por fecha_publicacion de forma descendente, luego por titulo de forma ascendente. El signo negativo en frente de "-fecha_publicacion"

indica orden *descendiente*. Si el - está ausente se asume un orden ascendente. Para ordenar aleatoriamente, usa "?", así:

```
>>> Entrada.objects.order_by('?')
```

distinct()

Retorna un nuevo QuerySet que usa SELECT DISTINCT en su consulta SQL. Esto elimina filas duplicadas en el resultado de la misma.

Por omisión, un QuerySet no eliminará filas duplicadas. En la práctica esto raramente es un problema porque consultas simples como Blog.objects.all() no introducen la posibilidad de registros duplicados.

Sin embargo, si tu consulta abarca múltiples tablas, es posible obtener resultados duplicados cuando un QuerySet es evaluado. Esos son los casos en los que usarías distinct().

values(*fields)

Retorna un QuerySet especial que evalúa a una lista de diccionarios en lugar de objetos instancia de modelo. Cada uno de esos diccionarios representa un objeto, con las las claves en correspondencia con el nombre de los atributos de los objetos modelo:

```
# Esta lista contiene un objeto Blog.
>>> Blog.objects.filter(nombre__startswith='Beatles')
[Beatles Blog]

# Esta lista contiene un diccionario.
>>> Blog.objects.filter(nombre__startswith='Beatles').values()
[{'id': 1, 'nombre': 'Beatles Blog', 'etiqueta': 'Las últimas novedades de los Beatles.'}]
```

values() puede recibir argumentos posicionales opcionales, *campos, los cuales especifican los nombres de campos a los cuales debe limitarse el SELECT. Si especificas los campos, cada diccionario contendrá solamente las claves/valores de campos para los campos que especifiques.

Si no especificas los campos, cada diccionario contendrá una clave y un valor para todos los campos en la table de base de datos:

```
>>> Blog.objects.values()
[{'id': 1, 'nombre': 'Beatles Blog', 'etiqueta': 'All the latest Beatles news.'}],
>>> Blog.objects.values('id', 'nombre')
[{'id': 1, 'nombre': 'Beatles Blog'}]
```

Este método es útil cuando sabes de antemano que solo vas a necesitar valores de un pequeño número de los campos disponibles y no necesitarás la funcionalidad de un objeto instancia de modelo. Es más eficiente el seleccionar solamente los campos que necesitas usar.

dates(field, kind, order)

Retorna un QuerySet especial que evalúa a una lista de objetos datetime.datetime que representan todas las fechas disponibles de un cierto tipo en el contenido de la QuerySet.

El argumento campo debe ser el nombre de un DateField o de un DateTimeField de tu modelo. El argumento tipo debe ser ya sea year, month o day. Cada objeto datetime.datetime en la lista de resultados es truncado de acuerdo al tipo provisto:

- **"year"**: retorna una lista de todos los valores de años distintos entre sí para el campo.
- **"month":** retorna una lista de todos los valores de años/mes distintos entre sí para el campo.
- **"day":** retorna una lista de todos los valores de años/mes/día distintos entre sí para el campo.
- **"orden":** cuyo valor por omisión es 'ASC', debe ser 'ASC' o 'DESC'. El mismo especifica cómo ordenar los resultados.

Aquí tenemos algunos ejemplos:

```
>>> Entrada.objects.dates('fecha_publicacion', 'year')
[datetime.datetime(2005, 1, 1)]
>>> Entrada.objects.dates('fecha_publicacion', 'month')
[datetime.datetime(2005, 2, 1), datetime.datetime(2005, 3, 1)]

>>> Entrada.objects.dates('fecha_publicacion', 'day')
[datetime.datetime(2005, 2, 20), datetime.datetime(2005, 3, 20)]

>>> Entrada.objects.dates('fecha_publicacion', 'day', order='DESC')
[datetime.datetime(2005, 3, 20), datetime.datetime(2005, 2, 20)]

>>> Entrada.objects.filter(titulo__contains='Lennon').dates('fecha_publicacion', 'day')
[datetime.datetime(2005, 3, 20)]
```

select_related()

Retorna un QuerySet que seguirá automáticamente relaciones de clave foránea, seleccionando esos datos adicionales de objetos relacionados cuando ejecuta su consulta. Esto contribuye a la mejora de rendimiento que resulta en consultas (aveces mucho) más grandes pero significan que el uso posterior de relaciones de clave foránea no requerirán consultas a la base de datos.

Los siguientes ejemplos ilustran la diferencia entre búsquedas normales y búsquedas select_related().

Esta es una búsqueda normal:

```
# Consulta a la base de datos.
>>> e = Entrada.objects.get(id=1)

# Consulta nuevamente a la base de datos para obtener los objetos blog relacionados.
>>> b = e.blog
```

Esta es una búsqueda select_related:

```
# Consulta a la base de datos.
>>> e = Entrada.objects.select_related().get(id=1)
```

```
# No consulta la base de datos, porque e.blog ha sido precargada en la anterior consulta.
>>> b = e.blog
```

select_related() sigue claves foráneas tan lejos como le sea posible. Si tienes los siguientes modelos:

```
class Ciudad(models.Model):
    # ...

class Persona(models.Model):
    # ...
    ciudad_natal = models.ForeignKey(Ciudad)

class Libro(models.Model):
    # ...
    autor = models.ForeignKey(Persona)
```

Entonces una llamada a Libro.objects.select_related().get(id=4) colocará en el cache la Persona relacionada *y* la Ciudad relacionada:

```
>>>b = Libro.objects.select_related().get(id=4)
>>> p = b.autor         # No consulta la base de datos.
>>> c = p.ciudad_natal      # No consulta la base de datos.

>>> b = Libro.objects.get(id=4) # En este ejemplo no select_related()
>>> p = b.autor         # Consulta a la base de datos.
>>> c = p.ciudad_natal      # Consulta a la base de datos.
```

Nota que *select_related* no sigue claves foráneas que tienen asignado null=True.

Usualmente, el usar select_related() puede mejorar muchísimo el desempeño porque tu aplicación puede puede entonces evitar muchas llamadas a la base de datos. Sin embargo, en siuaciones con conjuntos de relaciones profundamente anidadas, select_related() puede en algunos casos terminar siguiendo "demasiadas" relaciones y puede generar consultas tan grandes que terminan siendo lentas.

extra()

A veces, el lenguaje de consulta de Django no puede expresar facilmente cláusulas WHERE complejas. Para estos casos extremos, Django provee un modificador de QuerySet llamado extra() – una forma de inyectar cláusulas especificas dentro del SQL generado por un QuerySet.

Por definición, estas consultas especiales pueden no ser portables entre los distintos motores de bases de datos (debido a que estás escribiendo código SQL explícito) y violan el principio DRY, así que deberías evitarlas de ser posible.

Se puede especificar uno o más de params, select, where, o tables. Ninguno de los argumentos es obligatorio, pero deberías indicar al menos uno.

El argumento select permite indicar campos adicionales en una cláusula de SELECT. Debe contener un diccionario que mapee nombres de atributo a cláusulas SQL que se utilizarán para calcular el atributo en cuestión:

```
>>> Entrada.objects.extra(select={'is_recent': "fecha_publicacion > '2006-01-01'"})
```

Como resultado, cada objeto Entrada tendrá en este caso un atributo adicional, is_recent, un booleano que representará si el atributo fecha_publicacion del entrada es mayor que el 1 de Enero de 2006.

El siguiente ejemplo es más avanzado; realiza una subconsulta para darle a cada objeto Blog resultante un atributo entrada_count, un entero que indica la cantidad de objetos Entrada asociados al blog:

```
>>> subq = 'SELECT COUNT(*) FROM blog_entrada WHERE blog_entrada.blog_id =
        blog_blog.id'
>>> Blog.objects.extra(select={'entrada_count': subq})
```

(En este caso en particular, estamos aprovechando el hecho de que la consulta ya contiene la tabla blog_blog en su cláusula FROM.)

También es posible definir cláusulas WHERE explícitas – quizás para realizar joins implícitos – usando el argumento where. Se puede agregar tablas manualmente a la cláusula FROM del SQL usando el argumento tables.

Tanto where como tables reciben una lista de cadenas. Todos los argumentos de where son unidos con AND a cualquier otro criterio de búsqueda:

```
>>> Entrada.objects.extra(where=['id IN (3, 4, 5, 20)'])
```

Los parámetros select y where antes descritos pueden utilizar los comodines normales para bases de datos en Python: '%s' para indicar parámetros que deberían ser escapados automáticamente por el motor de la base de datos. El argumento params es una lista de los parámetros que serán utilizados para realizar la sustitución:

```
>>> Entrada.objects.extra(where=['titulo=%s'], params=['Lennon'])
```

Siempre se debe utilizar params en vez de utilizar valores directamente en select o where ya que params asegura que los valores serán escapados correctamente de acuerdo con tu motor de base de datos particular.

Este es un ejemplo de lo que está incorrecto:

```
Entrada.objects.extra(where=["titulo='%s'" % nombre])
```

Este es un ejemplo de lo que es correcto:

```
Entrada.objects.extra(where=['titulo=%s'], params=[nombre])
```

Metodos de QuerySet que no devuelven un QuerySet

Los métodos de QuerySet que se describen a continuación evaluan el QuerySet y devuelven algo *que no es* un QuerySet – un objeto, un valor, o algo así.

get(**lookup)

Devuelve el objeto que matchee el parámetro de búsqueda provisto. El parámetro debe proveerse de la manera descripta en la sección "Patrones de búsqueda". Este método levanta AssertionError si más de un objecto concuerda con el patrón provisto.

Si no se encuentra ningún objeto que coincida con el patrón de búsqueda provisto get() levanta una excepción de DoesNotExist. Esta excepción es un atributo de la clase del modelo, por ejemplo:

```
>>> Entrada.objects.get(id='foo') # levanta Entrada.DoesNotExist
```

La excepción DoesNotExist hereda de la clase: django.core.exceptions.ObjectDoesNotExist, así que puedes protegerte de múltiples excepciones DoesNotExist:

```
>>> from django.core.exceptions import ObjectDoesNotExist
>>> try:
... e = Entrada.objects.get(id=3)
... b = Blog.objects.get(id=1)
... except ObjectDoesNotExist:
... print ("La entrada o el blog no existen.")
```

create(kwargs)**

Este método sirve para crear un objeto y guardarlo en un mismo paso. Te permite abreviar dos pasos comunes:

```
>>> p = Persona(first_nombre="Bruce", last_nombre="Springsteen")
>>> p.save()
```

en una sola línea:

```
>>> p = Persona.objects.create(first_nombre="Bruce", last_nombre="Springsteen")
```

get_or_create(kwargs)**

Este método sirve para buscar un objeto y crearlo si no existe. Devuelve una tupla (object, created), donde object es el objecto encontrado o creado, y created es un booleano que indica si el objeto fue creado.

Está pensado como un atajo para el caso de uso típico y es más que nada útil para scripts de importación de datos, por ejemplo:

```
try:
   obj = Persona.objects.get(nombre = 'John', apellido = 'Lennon')
except Persona.DoesNotExist:
   obj = Persona(nombre = 'John',  apellido='Lennon',
      fecha_nacimiento= date(1940, 10, 9))
   obj.save()
```

Este patrón se vuelve inmanejable a medida que aumenta el número de campos en el modelo. El ejemplo anterior puede ser escrito usando get_or_create así:

```
obj, created = Persona.objects.get_or_create(
   nombre = 'John',
   apellido = 'Lennon',
   defaults = {'fecha_nacimiento': date(1940, 10, 9)}
)
```

Cualquier argumento que se le pase a get_or_create() – *excepto* el argumento opcional defaults – será utilizado en una llamada a get(). Si se encuentra un objecto, get_or_create devolverá una tupla con ese objeto y False. Si *no* se encuentra un

objeto, get_or_create() instanciará y guardará un objeto nuevo, devolviendo una tupla con el nuevo objeto y True. El nuevo objeto será creado de acuerdo con el siguiente algoritmo:

```
defaults = kwargs.pop('defaults', {})
params = dict([(k, v) for k, v in kwargs.items() if '__' not in k])
params.update(defaults)
obj = self.model(**params)
obj.save()
```

Esto es, se comienza con los argumentos que no sean 'defaults' y que no contengan doble guión bajo (lo cual indicaría una búsqueda no exacta). Luego se le agrega el contenido de defaults, sobreescribiendo cualquier valor que ya estuviera asignado, y se usa el resultado como claves para el constructor del modelo.

Si el modelo tiene un campo llamado defaults y es necesario usarlo para una búsqueda exacta en get_or_create(), simplemente hay que utilizar 'defaults__exact' así:

```
Foo.objects.get_or_create(
   defaults__exact = 'bar',
   defaults={'defaults': 'baz'}
)
```

count()

Devuelve un entero representando el número de objetos en la base de datos que coincidan con el QuerySet. count() nunca levanta excepciones. He aquí un ejemplo:

```
# Returns the total number of entries in the database.
>>> Entrada.objects.count()
4

# Returns the number of entries whose titulo contains 'Lennon'
>>> Entrada.objects.filter(titulo__contains='Lennon').count()
1
```

count() en el fondo realiza un SELECT COUNT(*), así que deberías siempre utilizar count() en vez de cargar todos los registros en objetos Python y luego invocar len() sobre el resultado.

Dependiendo de la base de datos que estés utilizando (e.g., PostgreSQL o MySQL), count() podría devolver un entero largo en vez de un entero normal de Python. Esto es una característica particular de la implementación subyacente que no debería ser ningún problema en la vida real.

in_bulk(id_list)

Este método toma una lista de claves primarias y devuelve un diccionario que mapea cada clave primaria en una instancia con el ID dado, por ejemplo:

```
>>> Blog.objects.in_bulk([1])
{1: Beatles Blog}
>>> Blog.objects.in_bulk([1, 2])
```

```
{1: Beatles Blog, 2: Cheddar Talk}
>>> Blog.objects.in_bulk([])
{}
```

Si no se encuentra un objeto en la base para un ID en particular, este id no aparecerá en el diccionario resultante. Si le pasas una lista vacía a in_bulk(), obtendrás un diccionario vacío.

latest(field_name=None)

Devuelve el último objeto de la tabla, ordenados por fecha, utilizando el campo que se provea en el argumento field_nombre como fecha. Este ejemplo devuelve el Entrada más reciente en la tabla, de acuerdo con el campo fecha_publicacion:

```
>>> Entrada.objects.latest('fecha_publicacion')
```

Si el Meta de tu modelo especifica get_latest_by, se puede omitir el argumento field_nombre. Django utilizará el campo indicado en get_latest_by por defecto.

Al igual que get(), latest() levanta DoesNotExist si no existe un objeto con los parámetros provistos.

Patrones de búsqueda

Los patrones de búsqueda son la manera en que se especifica la carne de una cláusula WHERE de SQL. Consisten de argumentos de palabra clave para los métodos filter(), exclude() y get() de QuerySet.

Los parámetros básicos de búsqueda toman la forma de campo__tipodebusqueda=valor (notar el doble guión bajo). Por ejemplo:

```
>>> Entrada.objects.filter(fecha_publicacion__lte='2006-01-01')
```

Se traduce aproximadamente en el siguiente comando SQL:

```
SELECT * FROM blog_entrada WHERE fecha_publicacion <= '2006-01-01';
```

Si se suministra un argumento de palabra clave inválido, la función levantará una excepción de TypeError.

A continuación se listan los tipos de búsqueda que existen.

exact

Realiza una búsqueda por coincidencias exactas:

```
>>> Entrada.objects.get(titulo__exact="Los Beatles")
```

Esto busca objetos que tengan en el campo titulo la frase exacta “Man bites dog”.

Si no se suministra un tipo de búsqueda – O sea, si tu argumento de palabra clave no contiene un doble guión bajo – el tipo de búsqueda se asume como exact.

Por ejemplo, las siguientes dos sentencias son equivalentes:

```
>>> Blog.objects.get(id__exact=14) # De forma explicita
>>> Blog.objects.get(id=14) # __exact  esta implicito
```

Esto es por conveniencia, dado que las búsquedas con tipo de búsqueda exact son las más frecuentes.

iexact

Realiza una búsqueda por coincidencias exactas sin distinguir mayúsculas de minúsculas:

```
>>> Blog.objects.get(nombre__iexact='beatles blog')
```

Esta consulta traerá objetos con nombre 'Beatles Blog', 'beatles blog', 'BeAtLes BLoG', etcétera.

contains

Realiza una búsqueda de subcadenas, distinguiendo mayúsculas y minúsculas:

```
>>>Entrada.objects.get(titulo__contains='Lennon')
```

Esto coincidirá con el titular 'Today Lennon honored' pero no con 'today lennon honored'.
SQLite no admite sentencias LIKE distinguiendo mayúsculas y minúsculas; cuando se utiliza SQLite, contains se comporta como icontains.

ESCAPADO DE PORCIENTO Y GUIÓN BAJO EN SENTENCIAS LIKE

Los patrones de búsqueda que resulten en sentencias SQL LIKE (iexact, contains, icontains, startswith, istartswith, endswith, y iendswith) escaparán automáticamente los dos caracteres especiales utilizados en sentencias LIKE – el porciento y el guión bajo. (En una sentencia LIKE, el símbolo de porciento indica una secuencia de caracteres cualesquiera, y el guión bajo indica un solo caracter cualquiera).

Esto significa que las cosas deberían funcionar de manera intuitiva, porque la abstracción funciona bien. Por ejemplo, para obtener todos las Entradas que contengan un símbolo de porciento, simplemente hace falta utilizar el símbolo de porcentaje como cualquier otro carácter:

```
Entrada.objects.filter(titulo__contains='%')
```

Django se hace cargo del escapado. El SQL resultante será algo similar a esto:

```
SELECT ... WHERE titulo LIKE '%\%%';
```

Lo mismo vale para el guión bajo. Tanto el símbolo de porcentaje como el guión bajo se deberían manejar de manera transparente.

icontains

Realiza una búsqueda de subcadenas, sin distinguir mayúsculas y minúsculas:

```
>>> Entrada.objects.get(titulo__icontains='Lennon')
```

A diferencia de contains, icontains *sí* trerá today lennon honored.

gt, gte, lt, y lte

Estos representan los operadores de mayor a, mayor o igual a, menor a, y menor o igual a, respectivamente:

```
>>> Entrada.objects.filter(id__gt=4)
>>> Entrada.objects.filter(id__lt=15)
>>> Entrada.objects.filter(id__gte=1)
```

Estas consultas devuelven cualquier objeto con un ID mayor a 4, un ID menor a 15, y un ID mayor o igual a 1, respectivamente.

Por lo general estos operadores se utilizarán con campos numéricos. Se debe tener cuidado con los campos de caracteres, ya que el orden no siempre es el que uno se esperaría (i.e., la cadena "4" resulta ser *mayor* que la cadena "10").

in

Aplica un filtro para encontrar valores en una lista dada:

```
Entrada.objects.filter(id__in=[1, 3, 4])
```

Esto devolverá todos los objetos que tengan un ID de 1, 3 o 4.

startswith

Busca coincidencias de prefijos distinguiendo mayúsculas y minúsculas:

```
>>> Entrada.objects.filter(titulo__startswith='Will')
```

Esto encontrará los titulares "Will he run?" y "Willbur nombred judge", pero no "Who is Will?" o "will found in crypt".

istartswith

Realiza una búsqueda por prefijos, sin distinguir mayúsculas y minúsculas:

```
>>> Entrada.objects.filter(titulo__istartswith='will')
```

Esto devolverá los titulares "Will he run?", "Willbur nombred judge", y "will found in crypt", pero no "Who is Will?"

endswith y iendswith

Realiza búsqueda de sufijos, distinguiendo y sin distinguir mayúsculas de minúsculas, respectivamente:

```
>>> Entrada.objects.filter(titulo__endswith='cats')
>>> Entrada.objects.filter(titulo__iendswith='cats')
```

range

Realiza una búsqueda por rango:

```
>>> start_date = datetime.date(2005, 1, 1)
>>> end_date = datetime.date(2005, 3, 31)
>>> Entrada.objects.filter(fecha_publicacion__range=(start_date, end_date))
```

Se puede utilizar range en cualquier lugar donde podrías utilizar BETWEEN en SQL – para fechas, números, e incluso cadenas de caracteres.

year, month, and day

Para campos date y datetime, realiza búsqueda exacta por año, mes o día:

```
# Búsqueda por año
>>>Entrada.objects.filter(fecha_publicacion__year=2005)

# Búsqueda por mes -- toma enteros
>>> Entrada.objects.filter(fecha_publicacion__month=12)

# Búsqueda por día
>>> Entrada.objects.filter(fecha_publicacion__day=3)

# Combinación: devuelve todas las entradas de Navidad de cualquier año
>>> Entrada.objects.filter(fecha_publicacion__month=12, fecha_publicacion__day=25)
```

isnull

Toma valores True o False, que corresponderán a consultas SQL de IS NULL``y ``IS NOT NULL, respectivamente:

```
>>> Entrada.objects.filter(fecha_publicacion__isnull=True)
```

__isnull=True vs. __exact=None

Hay una diferencia importante entre __isnull=True y __exact=None. __exact=None *siempre* devolverá como resultado un conjunto vacío, ya que SQL requiere que ningún valor sea igual a NULL. __isnull determina si el campo actualmente contiene un valor NULL sin realizar la comparación.

search

Un booleano que realiza búsquedas full-text, que aprovecha el indexado full-text. Esto es como contains pero significativamente más rápido debido al indexado full-text.

Nótese que este tipo de búsqueda sólo está disponible en MySQL y requiere de manipulación directa de la base de datos para agregar el índice full-text.

El patrón de búsqueda pk

Por conveniencia, Django provee un patrón de búsqueda pk, que realiza una búsqueda sobre la clave primaria del modelo (pk por primary key, del inglés).

En el modelo de ejemplo Blog, la clave primaria es el campo id, así que estas sentencias serían equivalentes:

```
>>> Blog.objects.get(id__exact=14) # Forma explícita
>>> Blog.objects.get(id=14) # __exact implícito
>>> Blog.objects.get(pk=14) # pk implica id__exact
```

El uso de pk no se limita a búsquedas __exact, cualquier patrón de búsqueda puede ser combinado con pk para realizar una búsqueda sobre la clave primaria de un modelo:

```
# Buscar entradas en blogs con id 1, 4, o 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Buscar entradas en blogs con id > 14
>>> Blog.objects.filter(pk__gt=14)
```

Las búsquedas por pk también funcionan con joins. Por ejemplo, estas tres sentencias son equivalentes:

```
>>> Entrada.objects.filter(blog__id__exact=3) # Forma explícita
>>> Entrada.objects.filter(blog__id=3) # __exact implícito
>>> Entrada.objects.filter(blog__pk=3) # __pk implica __id__exact
```

Búsquedas complejas con Objetos Q

Los argumentos de palabras clave en las búsquedas – en filter() por ejemplo – son unidos con AND. Si necesitas realizar búsquedas más complejas (e.g., búsquedas con sentencias OR), puedes utilizar objetos Q.

Un objeto Q (django.db.models.Q) es un objeto que se utiliza para encapsular una colección de argumentos de palabra clave. Estos argumentos de palabra clave son especificados como se indica en la sección “Patrones de búsqueda”.

Por ejemplo, este objeto Q encapsula una consulta con un único LIKE:

```
Q(question__startswith='What')
```

Los objetos Q pueden ser combinados utilizando los operadores & y |. Cuando se utiliza un operador sobre dos objetos, se obtiene un nuevo objeto Q. Por ejemplo, un OR de dos consultas question__startswith sería:

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

Esto será equivalente a la siguiente cláusula WHERE en SQL:

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

Puede componer sentencias de complejidad arbitraria combinando objetos Q con los operadores & y |. También se pueden utilizar paréntesis para agrupar.

Cualquier función de búsqueda que tome argumentos de palabra clave (e.g., filter(), exclude(), get()) puede recibir también uno o más objetos Q como argumento posicional (no nombrado). Si se proveen multiples objetos Q como argumentos a una función de búsqueda, los argumentos serán unidos con AND, por ejemplo:

```
Poll.objects.get(
Q(question__startswith='Who'),
Q(fecha_publicacion=date(2005, 5, 2)) | Q(fecha_publicacion=date(2005, 5, 6))
)
```

Se traduce aproximadamente en la siguiente sentencia SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
   AND (fecha_publicacion = '2005-05-02' OR fecha_publicacion = '2005-05-06')
```

Las funciones de búsqueda pueden además mezclar el uso de objetos Q y de argumentos de palabra clave. Todos los argumentos provistos a una función de búsqueda (sean argumentos de palabra clave u objetos Q) son unidos con AND. Sin embargo, si se provee un objeto Q debe preceder la definición de todos los argumentos de palabra clave. Por ejemplo, lo siguiente:

```
Poll.objects.get(
   Q(fecha_publicacion=date(2005, 5, 2)) | Q(fecha_publicacion=date(2005, 5, 6)),
      question__startswith='Who')
```

Es una consulta válida, equivalente al ejemplo anterior, pero esto:

```
# CONSULTA INVALIDA
Poll.objects.get(
   question__startswith='Who',
   Q(fecha_publicacion=date(2005, 5, 2)) | Q(fecha_publicacion=date(2005, 5, 6)))
```

No es válido.

Hay algunos ejemplos disponibles online en http://www.djangoproject.com/documentation/ .

Objetos Relacionados

Cuando defines una relación en un modelo (i.e. un ForeignKey, OneToOneField, or ManyToManyField), las instancias de ese modelo tendrán una API conveniente para acceder a estos objetos relacionados.

Por ejemplo, si e es un objeto Entrada, puede acceder a su Blog asociado accediendo al atributo blog, esto es e.blog.

Django también crea una API para acceder al "otro" lado de la relación – el vínculo del modelo relacionado al modelo que define la relación. Por ejemplo, si b es un objeto Blog, tiene acceso a la lista de todos los objetos Entrada a través del atributo entrada_set: b.entrada_set.all().

Todos los ejemplos en esta sección utilizan los modelos de ejemplo Blog, Autor y Entrada que se definen al principio de esta sección.

Consultas Que Cruzan Relaciones

Django ofrece un mecanismo poderoso e intuitivo para "seguir" relaciones cuando se realizan búsquedas, haciéndose cargo de los JOINs de SQL de manera automática. Para cruzar una relación simplemente hace falta utilizar el nombre de campo de los campos relacionados entre modelos, separados por dos guiones bajos, hasta que llegues al campo que necesitabas.

Este ejemplo busca todos los objetos Entrada que tengan un Blog cuyo nombre sea 'Beatles Blog':

```
>>> Entrada.objects.filter(blog__nombre__exact='Beatles Blog')
```

Este camino puede ser tan largo como quieras.
También Funciona en la otra dirección. Para referirse a una relación "inversa", simplemente hay que utilizar el nombre en minúsculas del modelo.
Este ejemplo busca todos los objetos Blog que tengan al menos un Entrada cuyo título contenga 'Lennon':

```
>>> Blog.objects.filter(entrada__titulo__contains='Lennon')
```

Relaciones de Clave Foránea

Si un modelo contiene un ForeignKey, las instancias de ese modelo tendrán acceso al objeto relacionado (foráneo) a través de un simple atributo del modelo, por ejemplo:

```
e = Entrada.objects.get(id=2)
e.blog # Devuelve el objeto Blog relacionado
```

Se puede acceder y asignar el valor de la clave foránea vía el atributo. Como sería de esperar, los cambios a la clave foránea no se guardan en el modelo hasta que invoques el método save(), por ejemplo:

```
e = Entrada.objects.get(id=2)
e.blog = some_blog
e.save()
```

Si un campo ForeignKey tiene la opción null=True seteada (i.e. permite valores NULL), se le puede asignar None:

```
e = Entrada.objects.get(id=2)
e.blog = None
e.save() # "UPDATE blog_entrada SET blog_id = NULL ...;"
```

El acceso a relaciones uno-a-muchos se almacena la primera vez que se accede al objeto relacionado. Cualquier acceso subsiguiente a la clave foránea del mismo objeto son cacheadas, por ejemplo:

```
e = Entrada.objects.get(id=2)
print (e.blog )# Busca el Blog asociado en la base de datos.
print (e.blog) # No va a la base de datos; usa la versión cacheada.
```

Notar que el método de QuerySet select_related() busca inmediatamente todos los objetos de relaciones uno-a-muchos de la instancia:

```
e = Entrada.objects.select_related().get(id=2)
print (e.blog) # No va a la base de datos; usa la versión cacheada.
print (e.blog) # No va a la base de datos; usa la versión cacheada.
```

select_related() está documentada en la sección "Métodos de consulta que retornan nuevos QuerySets".

Relaciones de Clave Foreánea "Inversas"

Las relaciones de clave foránea son automáticamente simétricas – se infiere una relación inversa de la presencia de un campo ForeignKey que apunte a otro modelo.

Si un modelo tiene una ForeignKey, las instancias del modelo de la clave foránea tendrán acceso a un Manager que devuelve todas las instancias del primer modelo. Por defecto, este Manager se llama FOO_set, donde FOO es el nombre modelo que contiene la clave foránea, todo en minúsculas. Este Manager devuelve QuerySets, que pueden ser filtradas y manipuladas como se describe en la sección "Recuperando objetos".

Aquí se muestra un ejemplo:

```
b = Blog.objects.get(id=1)
b.entrada_set.all() # Encontrar todos los objetos Entrada relacionados a b.

# b.entrada_set es un Manager que devuelve QuerySets.
b.entrada_set.filter(titulo__contains='Lennon')
b.entrada_set.count()
```

Se puede cambiar el nombre del atributo FOO_set indicando el parámetro related_name en la definición del ForeignKey(). Por ejemplo, si el modelo Entrada fuera cambiado por blog = ForeignKey(Blog, related_name='entradas'), el ejemplo anterior pasaría a ser así:

```
b = Blog.objects.get(id=1)
b.entradas.all() # Encontrar todos los objetos Entrada relacionados a b.

# b.entries es un Manager que devuelve QuerySets.
b.entradas.filter(titulo__contains='Lennon')
b.entradas.count()
```

No se puede acceder al Manager de ForeignKey inverso desde la clase misma; debe ser accedido desde una instancia:

```
Blog.entrada_set # Raises AttributeError: "Manager must be accessed via instance".
```

Además de los metodos de QuerySet definidos en la sección "Recuperando Objetos", el Manager de ForeignKey tiene los siguientes métodos adicionales:

- **add(obj1, obj2, ...):** Agrega los objetos del modelo indicado al conjunto de objetos relacionados, por ejemplo:

  ```
  b = Blog.objects.get(id=1)
  e = Entrada.objects.get(id=234)
  b.entrada_set.add(e) # Associates Entrada e with Blog b.
  ```

- **create(**kwargs):** Crea un nuevo objeto, lo guarda, y lo deja en el conjunto de objetos relacionados. Devuelve el objeto recién creado:

```
b = Blog.objects.get(id=1)
e = b.entrada_set.create(titulo='Hello', texto='Hi',
    fecha_publicacion=datetime.date(2005, 1, 1))
# No hace falta llamar a e.save() acá -- ya ha sido guardado
```

Esto es equivalente a (pero más simple que) lo siguiente:

```
>>>b = Blog.objects.get(id=1)
>>>e = Entrada(blog=b, titulo='Hello', texto='Hi',
        fecha_publicacion=datetime.date(2005, 1, 1))
>>> e.save()
```

Nota que no es necesario especificar el argumento de palabra clave correspondiente al modelo que define la relación. En el ejemplo anterior, no le pasamos el parámetro blog` a create(). Django deduce que el campo Entrada del nuevo blog debería ser b.

- **remove(obj1, obj2, ...):** Quita los objetos indicados del conjunto de objetos relacionados:

```
b = Blog.objects.get(id=1)
e = Entrada.objects.get(id=234)
b.entrada_set.remove(e) # Desasocia la Entrada e del Blog b.
```

Para evitar inconsistencias en la base de datos, este método sólo existe para objetos ForeignKey donde null=True. Si el campo relacionado no puede pasar ser None (NULL), entonces un objeto no puede ser quitado de una relación sin ser agregado a otra. En el ejemplo anterior, al quitar a *e* de *b.entrada_set()* es equivalente a hacer *e.blog = None*, y dado que la definición del campo ForeignKey blog (en el modelo Entrada) no indica null=True, esto es una acción inválida.

- **clear():** Quita todos los objetos del conjunto de objetos relacionados:

```
b = Blog.objects.get(id=1)
b.entrada_set.clear()
```

Nota: que esto no borra los objetos relacionados -- simplemente los desasocia.

Al igual que remove(), clear solo está disponible para campos

ForeignKey donde null=True.

Para asignar todos los miembros de un conjunto relacionado en un solo paso, simplemente se le asigna al conjunto un objeto iterable, por ejemplo:

```
b = Blog.objects.get(id=1)
b.entrada_set = [e1, e2]
```

Si el método clear() está definido, todos los objetos pre-existentes serán quitados del entrada_set antes de que todos los objetos en el iterable (en este caso, la lista) sean agregados al conjunto. Si el método clear() *no* está disponible, todos los objetos del iterable son agregados al conjunto sin quitar antes los objetos pre-existentes.

Todas las operaciones "inversas" definidas en esta sección tienen efectos inmediatos en la base de datos. Toda creación, borradura y agregado son inmediata y automáticamente grabados en la base de datos.

Relaciones muchos-a-muchos

Ambos extremos de las relaciones muchos-a-muchos obtienen una API de acceso automáticamente. La API funciona igual que las funciones "inversas" de las relaciones uno-a-muchos (descriptas en la sección anterior).

La única diferencia es el nombrado de los atributos: el modelo que define el campo ManyToManyField usa el nombre del atributo del campo mismo, mientras que el modelo "inverso" utiliza el nombre del modelo original, en minúsculas, con el sufijo '_set' (tal como lo hacen las relaciones uno-a-muchos).

Un ejemplo de lo anterior lo hará más fácil de entender:

```
e = Entrada.objects.get(id=3)
e.autores.all() # Devuelve todos los objetos Autor para este Entrada.
e.autores.count()
e.autores.filter(nombre__contains='John')

a = Autor.objects.get(id=5)
a.entrada_set.all() # Devuelve todos los obejtos Entrada para este Autor.
```

Al igual que los campos ForeignKey, los ManyToManyField pueden indicar un related_name. En el ejemplo anterior, si el campo ManyToManyField en el modelo Entrada indicara related_name='entries', cualquier instancia de Autor tendría un atributo entries en vez de entrada_set.

¿Cómo son posibles las relaciones inversas?

El mapeador objeto-relacional requiere que definas relaciones en ambos extremos. Los desarrolladores Django creen que esto es una violación del principio DRY (Don't Repeat Yourself), así que Django sólo te exige que definas la relación en uno de los extremos. ¿Pero cómo es esto posible, dado que una clase modelo no sabe qué otros modelos se relacionan con él hasta que los otros modelos sean cargados?

La respuesta yace en la variable INSTALLED_APPS. La primera vez que se carga cualquier modelo, Django itera sobre todos los modelos en INSTALLED_APPS y crea las relaciones inversas en memoria como sea necesario. Esencialmente, una de las funciones de INSTALLES_APPS es indicarle a Django el dominio completo de modelos que se utiliza.

Consultas que Abarcan Objetos Relacionados

Las consultas que involucran objetos relacionados siguen las mismas reglas que las consultas que involucran campos normales. Cuando se indica el valor que se requiere

en una búsqueda, se puede utilizar tanto una instancia del modelo o bien el valor de la clave primaria del objeto.

Por ejemplo, si b es un objeto Blog con id=5, las tres siguientes consultas son idénticas:

```
Entrada.objects.filter(blog=b) # Consulta usando un objeto de una instancia.
Entrada.objects.filter(blog=b.id) # Consulta usando el id de una instancia.
Entrada.objects.filter(blog=5) # Consulta usando un id directamente
```

Borrando Objetos

El métodos para borrar se llama delete(). Este método inmediatamente borra el objeto y no tiene ningún valor de retorno:

```
e.delete()
```

También se puede borrar objetos en grupo. Todo objeto QuerySet tiene un método delete() que borra todos los miembros de ese QuerySet. Por ejemplo, esto borra todos los objetos Entrada que tengan un año de fecha_publicacion igual a 2005:

```
Entrada.objects.filter(fecha_publicacion__year=2005).delete()
```

Cuando Django borra un objeto, emula el comportamiento de la restricción de SQL ON DELETE CASCADE – en otras palabras, todos los objetos que tengan una clave foránea que apunte al objeto que está siendo borrado serán borrados también, por ejemplo:

```
b = Blog.objects.get(pk=1)
# Esto borra el Blog y todos sus objetos Entrada.
b.delete()
```

Nota que delete() es el único método de QuerySet que no está expuesto en el Manager mismo. Esto es un mecanismo de seguridad para evitar que accidentalmente solicites Entrada.objects.delete() y borres *todos* los Entrada. Si *realmente* quieres borrar todos los objetos, hay que pedirlo explícitamente al conjunto completo de objetos:

```
Entrada.objects.all().delete()
```

Métodos de Instancia Adicionales

Además de save() y delete(), un objeto modelo puede tener cualquiera o todos de los siguientes métodos.

get_FOO_display()

Por cada campo que indica la opción choices, el objeto tendrá un método get_FOO_display(), donde FOO es el nombre del campo. Este método devuelve el valor "legible" del campo. Por ejemplo, en el siguiente modelo:

```
GENDER_CHOICES = (
('M', 'Masaculino'),
('F', 'Femenino'),
)

class Persona(models.Model):
    nombre = models.CharField(max_length=20)
    genero = models.CharField(max_length=1, choices=GENDER_CHOICES)

    Cada instancia de Persona tendrá un método get_genero_display:

>>> p = Persona(nombre='John', genero='M')
>>> p.save()
>>> p.genero
'M'
>>> p.get_genero_display()
'Masculino'
```

get_next_by_FOO(**kwargs) y get_previous_by_FOO(**kwargs)

Por cada campo DateField y DateTimeField que no tenga null=True, el objeto tendrá dos métodos get_next_by_FOO() y get_previous_by_FOO(), donde FOO es el nombre del campo. Estos métodos devuelven el objeto siguiente y anterior en orden cronológico respecto del campo en cuestión, respectivamente, levantando la excepción DoesNotExist cuando no exista tal objeto.

Ambos métodos aceptan argumentos de palabra clave opcionales, que deberían ser de la forma descripta en la sección "Patrones de búsqueda".

Notar que en el caso de valores de fecha idénticos, estos métodos utilizarán el ID como un chequeo secundario. Esto garantiza que no se saltearán registros ni aparecerán duplicados. Hay un ejemplo completo en los ejemplos de la API de búsqueda, en http://www.djangoproject.com/documentation.

get_FOO_filename()

Todo campo FileField le dará al objeto un método get_FOO_filename(), donde FOO es el nombre del campo. Esto devuelve el nombre de archivo completo en el sistema de archivos, de acuerdo con la variable MEDIA_ROOT.

Nota que el campo ImageField es técnicamente una subclase de FileField, así que todo modelo que tenga un campo ImageField obtendrá también este método.

get_FOO_url()

Por todo campo FileField el objeto tendrá un método get_FOO_url(), donde FOO es el nombre del campo. Este método devuelve la URL al archivo, de acuerdo con tu variable MEDIA_URL.

Si esta variable está vacía, el método devolverá una cadena vacía.

get_FOO_size()

Por cada campo FileField el objeto tendrá un método get_FOO_size(), donde FOO es el nombre del campo. Este método devuelve el tamaño del archivo, en bytes. (La implementación de este método utiliza os.path.getsize.)

save_FOO_file(filename, raw_contents)

Por cada campo FileField, el objeto tendrá un método save_FOO_file(), donde FOO es el nombre del campo. Este método guarda el archivo en el sistema de archivos, utilizando el nombre dado. Si un archivo con el nombre dado ya existe, Django le agrega guiones bajos al final del nombre de archivo (pero antes de la extensión) hasta que el nombre de archivos esté disponible.

get_FOO_height() and get_FOO_width()

Por cada campo ImageField, el objeto obtendrá dos métodos, get_FOO_height() y get_FOO_width(), donde FOO es el nombre del campo. Estos métodos devuelven el alto y el ancho (respectivamente) de la imagen, en pixeles, como un entero.

Atajos (Shortcuts)

A medida que desarrolles tus vistas, descubrirás una serie de modismos en la manera de utilizar la API de la base de datos. Django codifica algunos de estos modismos como atajos que pueden ser utilizados para simplificar el proceso de escribir vistas. Estas funciones se pueden hallar en el módulo django.shortcuts.

get_object_or_404()

Un modismo frecuente es llamar a get() y levantar un Http404 si el objeto no existe. Este modismo es capturado en la función get_object_or_404(). Esta función toma un modelo Django como su primer argumento, y una cantidad arbitraria de argumentos de palabra clave, que le pasa al método get() del Manager por defecto del modelo. Luego levanta un Http404 si el objeto no existe, por ejemplo:

```
# Obtiene una Entrada con una clave primaria 3
e = get_object_or_404(Entrada, pk=3)
```

Cuando se le pasa un modelo a esta función, se utiliza el Manager por defecto para ejecutar la consulta get() subyacente. Si no quieres que se utilice el manager por defecto, o si quiere buscar en una lista de objetos relacionados, se le puede pasar a get_object_or_404() un objeto Manager en vez:

```
# Obtiene el autor del blog con una intancia de e,   con el nombre de 'Fred'
a = get_object_or_404(e.autores, nombre='Fred')

# Usa un manager personalizado 'entradas_recientes' en la búsqueda de  una entrada
# con  una clave primaria 3
e = get_object_or_404(Entrada. entradas_recientes, pk=3)
```

get_list_or_404()

get_list_or_404() se comporta igual que get_object_or_404(), salvo porque llama a filter() en vez de a get(). Levanta un Http404 si la lista resulta vacía.

Utilizando SQL Crudo

Si te encuentras necesitando escribir una consulta SQL que es demasiado compleja para manejarlo con el mapeador de base de datos de Django, todavía puede optar por escribir la sentencia directamente en SQL crudo.

La forma preferida para hacer esto es dándole a tu modelo métodos personalizados o métodos de Manager personalizados que realicen las consultas. Aunque no exista ningún requisito en Django que *exija* que las consultas a la base de datos vivan en la capa del modelo, esta implementación pone a toda tu lógica de acceso a los datos en un mismo lugar, lo cual es una idea astuta desde el punto de vista de organización del código.

Finalmente, es importante notar que la capa de base de datos de Django es meramente una interfaz a tu base de datos. Puedes acceder a la base de datos utilizando otras herramientas, lenguajes de programación o frameworks de bases de datos – No hay nada específicamente de Django acerca de tu base de datos.

APÉNDICE C

Referencia de las vistas genéricas

El *capítulo 11* es una introducción a las vistas genéricas basadas en clases, pero pasa por alto algunos detalles importantes. Este apéndice describe todas las vistas genéricas, junto con las opciones que cada una de ellas puede aceptar. Antes de intentar entender este material de referencia es muy conveniente leer el *capítulo 11*. Tampoco viene mal un repaso a los modelos Libro, Editor y Autor definidos en dicho capítulo, ya que serán usados en los ejemplos incluidos en este apéndice.

Argumentos comunes a las vistas genéricas basadas en clases

La mayoría de las vistas genéricas basadas en clases, aceptan algunos argumentos que pueden modificar su comportamiento. Muchos de esos argumentos funcionan igual para la mayoría de las vistas (hay sus excepciones). La tabla C-1 describe algunos de estos argumentos comunes; cada vez que veas uno de estos argumentos en la lista de parámetros admitidos por una vista genérica, su comportamiento será tal y como se describe en esta tabla.

Argumento	Descripción
allow_empty	Un valor booleano que indica cómo debe comportarse la vista si no hay objetos disponibles. Si vale False y no hay objetos, la vista elevará un error 404 en vez de mostrar una página vacía. Su valor por defecto es Falsa.
context_object_name	El nombre de la variable principal en el contexto de la plantilla. Por defecto, es 'object'. Para las listas que utilizan más de objeto (por ejemplo, las vistas de listados o de archivos por fechas), se añade el sufijo '_list' al valor de este parámetro, así que si no se indica nada y la vista utiliza varios objetos, estos estarán accesibles mediante una variable llamada object_list.
context_processors	Es una lista de procesadores de contexto adicionales (además de los incluidos por el sistema), que se aplican a la plantilla de la vista.
model	El modelo del cual la vista mostrara los datos. Especificar model = Foo es tan efectivo como especificar queryset = Foo.objects.all()

mimetype	El tipo MIME a usar para el documento resultante. Por defecto utiliza el tipo definido en la variable de configuración DEFAULT_MIME_TYPE, cuyo valor inicial es text/html.
queryset	Un objeto de tipo QuerySet (por ejemplo, Autor.objects.all()) del cual se leerán los objetos a utilizar por la vista. En el apéndice C hay más información acerca de los objetos QuerySet. La mayoría de las vistas genéricas necesitan este argumento.
paginate_by	El número máximo de objetos para paginar permitidos en una plantilla (Disponible solo en listas de objetos).
template_name	El nombre completo de la plantilla a usar para representar la página. Este argumento se puede usar si queremos modificar el nombre que se genera automáticamente a partir del QuerySet.

Tabla C-1. *Argumentos comunes a la mayoría de vistas genéricas.*

Vistas genéricas basadas en clases-base

Dentro del módulo django.views.generic existen varias vistas sencillas que manejan unos cuantos problemas frecuentes: mostrar una plantilla que no necesita una vista lógica ***TemplateView***, hacer una redirección de una página ***RedirectView*** y la más importante de todas, la clase base maestra, de la cual todas las demás clases genéricas heredan llamada ***View.*** Estas tres clases implementan muchas de las funcionalidades necesitadas para crear vistas en Django, puedes pensar en ellas como `vistas padres que pueden usarse en sí mismas o heredando sus atributos dependiendo de la complejidad y las necesidades de tu proyecto.

Muchas de las características incorporadas en las vistas basadas en clases, heredan de otras clases o de varias clases usando mixins. Debido a que la cadena de herencia es muy importante, las clases ancestro están documentadas debajo del título de cada sección de sus ancestros (MRO). MRO es un acrónimo para el orden de resolución de un método.

Vista base: *View*

Clase: django.views.generic.base.View

La clase-base maestra de todas las vistas genéricas. Todas las vistas basadas en clases genéricas heredan de esta clase base.

Flujo de métodos:

1. dispatch()
2. http_method_not_allowed()
3. options()

Ejemplo:

Para crear el famoso hola mundo usando una clase genérica, creamos una vista *View* importándola del modulo *django.views.generic* de la siguiente forma:

views.py

```
from django.http import HttpResponse
from django.views.generic import View

class MiVista(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('¡Hola, Mundo!')
```

Luego la enlazamos directamente en la URL:

urls.py

```
from django.conf.urls import url
from aplicacion.views import MiVista

urlpatterns = [
    url(r'^hola/$', MiVista.as_view(), name='mi-vista'),
]
```

Atributos

http_method_names: La lista de nombre de métodos HTTP, que esa vista acepta son:

Default:

['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']

Métodos

- **classmethod as_view(**initkwargs):** Retorna una vista llamable que toma una petición y retorna una respuesta:

  response = MiVista.as_view()(request)

- **dispatch(request, *args, **kwargs):** La view la vista – el método que acepta un argumento request mas los argumentos pasados y devuelve una respuesta HTTP.

  La implementación predeterminada inspecciona el método HTTP y trata de delegarlo a el método que coincida con la petición HTTP; una petición GET será delegada a un método get(), una POST a un post() y así sucesivamente.

  Por omisión una petición a HEAD será delegada al método get(). Si necesitas manejar peticiones HEAD de diferentes formas usa GET para sobrescribir el método head().

- **http_method_not_allowed(request, *args, **kwargs):** Si la vista es llamada mediante un método HTTP no soportado, este método es llamado en su lugar.

La implementación predeterminada devuelve un HttpResponseNotAllowed con una lista de métodos permitidos en texto plano.

- **options(request, *args, **kwargs):** Maneja la respuesta a las peticiones para los verbos OPTIONS HTTP. Devuelve una lista de nombres de métodos HTTP permitidos para las vistas.

Renderizar una plantilla con *TemplateView*

Clase: django.views.generic.base.TemplateView

Renderiza una plantilla dada, con el contexto que contiene los parámetros capturados en la URL.

Ancestros (MRO)

Esta vista hereda métodos y atributos de las siguientes vistas:

- django.views.generic.base.TemplateResponseMixin
- django.views.generic.base.ContextMixin
- django.views.generic.base.View

Flujo de métodos:

1. dispatch()
2. http_method_not_allowed()
3. get_context_data()

Ejemplo:

Para mostrar una página de bienvenida que muestre los últimos 5 libros publicados en la base de datos, usamos la clase TemplateView directamente podríamos hacerlo así:

views.py

```
from django.views.generic.base import TemplateView
from biblioteca.models import Libro

class PaginaBienvenida(TemplateView):
    template_name = "bienvenida.html"

    def get_context_data(self, **kwargs):
        context = super(PaginaBienvenida, self).get_context_data(**kwargs)
        context['ultimos_libros'] = Libro.objects.all()[:5]
        return context
```

Después solo la enlazamos a su respectiva URL:

```
urls.py
from django.conf.urls import url
from biblioteca.views import PaginaBienvenida

urlpatterns = [
    url(r'^$', PaginaBienvenida.as_view(), name='bienvenidos'),
]
```

Atributos

Rellena (A través de la clase ContextMixin) con los argumentos clave, capturados del patrón URL que sirve la vista.

Redirigir a otra URL mediante *RedirectView*

Clase: django.views.generic.base.RedirectView

Esta vista redirige a otra URL.

La URL dada puede contener un formato de estilo tipo diccionario, que será intercalado contra los parámetros capturados en la URL. Ya que el intercalado de palabras claves se hace *siempre* (incluso si no se le pasan argumentos), por lo que cualquier carácter como "%" (un marcador de posición en Python) en la URL debe ser escrito como "%%" de modo que Python lo convierta en un simple signo de porcentaje en la salida.

Si la URL pasada como parámetro es None, Django retornará un mensaje de error 410 ("Gone" según el estándar HTTP).

Ancestros (MRO)

Esta vista hereda métodos y atributos de las siguientes vistas:

- django.views.generic.base.View

Flujo de métodos:

1. dispatch()
2. http_method_not_allowed()
3. get_redirect_url()

Ejemplo:

Supongamos que queremos redirecionar a nuestros usuarios a una página que actualiza un ficticio contador de libros, después de que visiten una página de detalles:

```
views.py
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView

from biblioteca.models import Libro

class RedirecionarContadorLibros(RedirectView):
    permanent = False
    query_string = True
```

```
    pattern_name = 'detalles-libro'

    def get_redirect_url(self, *args, **kwargs):
        libro = get_object_or_404(Libro, pk=kwargs['pk'])
        libro.update_counter()
        return super(RedirecionarContadorLibros, self).get_redirect_url(*args, **kwargs)
```

urls.py

```
from django.conf.urls import url
from django.views.generic.base import RedirectView

from biblioteca.views import RedirecionarContadorLibros, DetalleLibros
urlpatterns = [
    url(r'^contador/(?P<pk>[0-9]+)/$', RedirecionarContadorLibros.as_view(),
        name='contador-libros'),
    url(r'^detalles/(?P<pk>[0-9]+)/$', DetalleLibros.as_view(),
        name='detalles-libro'),
    url(r'^go-to-django/$', RedirectView.as_view(url='http://djangoproject.com'),
        name='go-to-django'),
]
```

Atributos

- **url** La URL a redirecionar, como una cadena o string. O None para lanzar un error 410(Gone).

- **pattern_name:** El nombre de el patrón URL para redireccionamiento. El redireccionamiento puede hacerse usando los mismos argumentos: args y kwargs que son pasados en la vistas.

- **Permanent: Indica** si el redireccionamiento debería ser permanente, La única diferencia aquí es el código del estatus HTTP que devuelve. Si es True, el redirecionamiento usara un código de estatus 301. Si es False, el código de estatus será 302. El valor predeterminado para permanent es True.

- **query_string: Indica** si se le pasa la cadena de consulta GET a la nueva localización. Si es True, la cadena de consulta es agregada a la URL. Si es False la cadena de consulta es descartada. El valor predeterminado para query_string es False.

Métodos

- ***get_redirect_url(*args, **kwargs)***: Construye la URL del objetivo, para el cambio de dirección.

  La implementación predeterminada usa el atributo url cuando comienza como una cadena y optimiza la expansión de los nombres de parámetros % capturados en la cadena, usando los nombres de grupos capturados en la URL.

Si la url no está establecido, get_redirect_url() trata de usar el inverso de pattern_name usando los valores capturados en la URL (usando nombres y nombres de grupos)

Si la petición de query_string, es agregada a la cadena de consulta para general la URL. La subclase puede implementar cualquier comportamiento que desee, mientras que el método devuelva un redireccionamiento listo para una cadena de una URL.

Vistas de listado/detalle

Las vistas genéricas basadas en clases de listados/detalle (que residen en el módulo django.views.generic) se encargan de la habitual tarea de mostrar una lista de elementos por un lado (el listado) y una vista individual para cada uno de los elementos (el detalle)

Listas de objetos: *ListView*

Clase: django.views.generic.list.ListView

Una página que representa una lista de objetos.

Mientras esta vista es ejecutada con self.object_list contiene una lista de objetos (usualmente, pero no necesariamente un queryset) sobre los que la vista está operando.

Ancestros (MRO)

Esta vista hereda métodos y atributos de las siguientes vistas:

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.list.BaseListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.base.View

Flujo de métodos:

1. dispatch()
2. http_method_not_allowed()
3. get_template_names()
4. get_queryset()
5. get_context_object_name()
6. get_context_data()
7. get()
8. render_to_response()

La clase ListView usa la clase BaseListView, al igual que las vistas que necesitan presentar una lista de objetos.

Clase: django.views.generic.list.BaseListView

Una vista base para mostrar una lista de objetos, No está pensada para ser usada directamente, pero puede usarse como una clase padre para *django.views.generic.list.ListView* otras vistas que representen una lista de objetos.

Ancestros (MRO)

Esta vista hereda métodos y atributos de las siguientes vistas:

- django.views.generic.list.MultipleObjectMixin
- django.views.generic.base.View

Métodos

get(*request*, **args*, ***kwargs*): Agrega object_list al contexto. Si el atributo allow_empty es True muestra una lista vacía. Si el atributo allow_empty es False lanza un error 404.

Ejemplo:

Si consideramos el objeto Autor tal y como se definió en el capítulo 5, podemos usar la vista ListView para obtener un listado sencillo de todos los autores usando la siguiente vista genérica (usando una clase) y su respectiva URLconf:

biblioteca/views.py

```
from django.views.generic import ListView
from biblioteca.models import Autor

# El único requerimiento es un queryset o modelo.
class ListaAutores(ListView):
model = Autor
```

biblioteca/urls.py

```
from django.conf.urls import url
from biblioteca.views import ListaAutores

# Enlazamos la vista usando el método as_view()
urlpatterns = [
    url(r'^autores/$', ListaAutores.as_view()),
]
```

templates/biblioteca/autor_list.html

```
<h1>Lista de Autores</h1>

<ul>
{% for autor in object_list %}
    <li><a href="{% url 'detalles-autores' autor.id %}">{{ autor.nombre }}
        {{ autor.apellidos }}</li>
{% empty %}
    <li>No hay autores registrados.</li>
{% endfor %}
</ul>
```

La vista ListView usa el método interno get_absolute_url() de el modelo para enlazar la url y la vista detallada de un objeto DetailView, de la siguiente forma(De lo contrario lanzara una excepción):

biblioteca/models.py

```
from django.core.urlresolvers import reverse
from django.db import models

# La clase que define al modelo autor
class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    # ...

    # Muestra los detalles del objeto
    def get_absolute_url(self):
        return reverse('detalles-autores', args=[self.pk])
```

Argumentos obligatorios

- **queryset:** Un QuerySet de los objetos a listar (Véase la tabla C-1) o un model: El modelo del cual la vista mostrara los datos, como en el ejemplo anterior, model = Autor es equivalente a usar un queryset = Autor.objects.all()

Argumentos opcionales

- **paginate_by:** (Paginación) es un número entero que especifica cuantos objetos se deben mostrar en cada página. Según se especifique en este parámetro, los resultados serán paginados, de forma que se distribuirán por varias páginas de resultado. La vista determinará que página de resultados debe mostrar o bien desde un parámetro page incluido en la URL (vía Get) o mediante una variable page especificada en el URLconf. En cualquiera de los dos casos, el índice comienza en cero. En la siguiente sección hay una nota sobre paginación donde se explica con un poco más de detalle este sistema.

- **template_name:** (Nombre de la plantilla) Si no se ha especificado el parámetro opcional template_name, la vista usará una plantilla llamada <app_label>/<model_name>_list.html. Tanto la etiqueta de la aplicación como la etiqueta del modelo se obtienen del parámetro queryset. La etiqueta de aplicación es el nombre de la aplicación en que se ha definido el modelo, y la etiqueta de modelo es el nombre, en minúsculas, de la clase del modelo.

  En el ejemplo anterior, tendríamos que el queryset sería Autor.objects.all(), por lo que la etiqueta de la aplicación será biblioteca y el nombre del modelo es autor. Con esos datos, el nombre de la plantilla a utilizar por defecto será biblioteca/autor_list.html.

- **contex_object_name**: El nombre del contexto a usar en las plantillas, si no se define uno, se usara el predeterminado que es: "*object_list*"

Además de los valores que se puedan haber definido en el contexto, la plantilla tendrá los siguientes valores:

- **object_list:** La lista de los objetos. El nombre de la variable viene determinado por el parámetro template_object_name, y vale 'object' por defecto. Si se definiera template_object_name como 'foo', el nombre de esta variable sería foo_list.

- **is_paginated:** Un valor booleano que indicará si los resultados serán paginados o no. Concretamente, valdrá False si el número de objetos disponibles es inferior o igual a paginate_by.

  Si los resultados están paginados, el contexto dispondrá también de estas variables:

- **results_per_page**: El número de objetos por página. (Su valor es el mismo que el del parámetro paginate_by).

- **has_next:** Un valor booleano indicando si hay una siguiente página.

- **has_previous:** Un valor booleano indicando si hay una página previa.

- **page:** El número de la página actual, siendo 1 la primera página.

- **next:** El número de la siguiente página. Incluso si no hubiera siguiente página, este valor seguirá siendo un número entero que apuntaría a una hipotética siguiente página. También utiliza un índice basado en 1, no en cero.

- ***previous:*** El número de la anterior página, usando un índice basado en 1, no en cero.

- ***pages:*** El número total de páginas.

- ***hits***: El número total de objetos en *todas* las páginas, no sólo en la actual.

Una nota sobre paginación:

Si se utiliza el parámetro paginate_by, Django paginará los resultados. Puedes indicar qué pagina visualizar usando dos métodos diferentes:

1. Usar un parámetro *page* en el URLconf.
2. Pasar el número de la página mediante un parámetro page en la URL.

En ambos casos, page es un índice basado en 1, lo que significa que la primera página siempre será la número 1, no la número 0.

Listas de objetos: *DetailView*

Clase: django.views.generic.detail.DetailView

Esta vista proporciona una representación individual de los "detalles" de un objeto. Cuando esta vista es ejecutada self.object contiene un objeto sobre el que la vista opera.

Ancestros (MRO)

Esta vista hereda métodos y atributos de las siguientes vistas:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

Flujo de métodos:

1. dispatch()
2. http_method_not_allowed()
3. get_template_names()
4. get_slug_field()
5. get_queryset()
6. get_object()
7. get_context_object_name()
8. get_context_data()
9. get()
10. render_to_response()

Ejemplo:

Siguiendo con el ejemplo anterior, podemos añadir una vista de detalle de cada autor modificando el URLconf y pasándole un contexto extra ahora, de la siguiente manera:

biblioteca/views.py

```
from django.views.generic.detail import DetailView
from django.utils import timezone

from biblioteca.models import Autor

class DetalleAutores(DetailView):
    model = Autor

    # Le agregamos un contexto extra 'ahora', que muestra la fecha actual.
    def get_context_data(self, **kwargs):
        context = super(DetalleAutores, self).get_context_data(**kwargs)
        context['ahora'] = timezone.now()
        return context
```

biblioteca/urls.py

```
from django.conf.urls import url
from biblioteca.views import DetalleAutores

urlpatterns = [
    url(r'^detalle/autores/(?P<pk>[0-9]+)/$', DetalleAutores.as_view(),
        name='detalles-autores' ),
]
```

templates/biblioteca/autor_detail.html

```
{% extends "base.html" %}
{% block content %}
<h1>{{ object.nombre }}{{ object.apellidos}}</h1>

<ul>
   <li>Email: {{ object.email }}</li>
   <li>Ultimo acceso: {{ object.ultimo_acceso }}</li>
   <li>Fecha: {{ ahora|date }}</li>
</ul>
{% endblock %}
```

Argumentos obligatorios

- **queryset:** Un QuerySet que será usado para localizar el objeto a mostrar o un model (véase la Tabla C-1).

- **object_id:** El valor de la clave primaria del objeto a mostrar. En el ejemplo anterior usamos pk para capturar la clave primaria del objeto en la URL, para pasársela a la clase vista o puedes usar:

  slug: La etiqueta o slug del objeto en cuestión. Si se usa este sistema de identificación, hay que emplear obligatoriamente el argumento slug_field (que se explica en la siguiente sección).

Argumentos opcionales

- **slug_field**: El nombre del atributo del objeto que contiene el slug. Es obligatorio si estás usando el argumento slug, y no se debe usar si estás usando el argumento object_id.

- **template_name_field***:* El nombre de un atributo del objeto cuyo valor se usará como el nombre de la plantilla a utilizar. De esta forma, puedes almacenar en tu objeto la plantilla a usar.

  En otras palabras, si tu objeto tiene un atributo 'the_template' que contiene la cadena de texto 'foo.html', y defines template_name_field para que valga 'the_template', entonces la vista genérica de este objeto usará como plantilla 'foo.html'.

  Si el atributo indicado por template_name_field no existe, se usaría el indicado por el argumento template_name. Es un mecanismo un poco enmarañado, pero puede ser de mucha ayuda en algunos casos.

Nombre de la plantilla

- Si no se especifican *template_name* ni *template_name_field,* se usará la plantilla <app_label>/<model_name>_detail.html, como el nombre de la plantilla.

Atributos de plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla tendrá los siguientes valores:

- **object***:* El objeto. El nombre de esta variable puede ser distinto si se ha especificado el argumento context_object_name, cuyo valor es 'object' por defecto. Si definimos context_object_name como 'foo', el nombre de la variable será foo.

Vistas genéricas para Crear/Modificar/Borrar

Clase: django.views.generic.dates

El módulo django.views.generic.edit, contiene una serie de funciones para crear, modificar y borrar objetos.

Las vistas son las siguientes:

- django.views.generic.edit.FormView
- django.views.generic.edit.CreateView
- django.views.generic.edit.UpdateView
- django.views.generic.edit.DeleteView

Todas estas vistas presenta formularios si se acceden con GET y realizan la operación solicitada (crear/modificar/borrar) si se acceden con POST.

Estas vistas tienen un concepto muy simple de la seguridad. Aunque aceptan un argumento llamado login_required, que restringe el acceso sólo a usuarios identificados, no hacen nada más. Por ejemplo, no comprueban que el usuario que está modificando un objeto sea el mismo usuario que lo creo, ni validarán ningún tipo de permisos.

En cualquier caso, la mayor parte de las veces se puede conseguir esta funcionalidad simplemente escribiendo un pequeño recubrimiento alrededor de la vista genérica. Para más información sobre esta técnica, véase el *capítulo 11*.

Mostrar formularios con: *FormView*

Clase: django.views.generic.edit.FormView

Una vista que muestra un formulario. Si existen errores vuelve a mostrar el formulario con los errores de validación; si esta tiene éxito redirecciona a la nueva URL.

Ancestros (MRO)

Esta vista hereda métodos y atributos de las siguientes vistas:

- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseFormView
- django.views.generic.edit.FormMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

Ejemplo:

Supongamos que queremos mostrar un sencillo formulario de contactos:

forms.py

```
from django import forms

class FormularioContactos(forms.Form):
    nombre = forms.CharField()
    mensaje = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # Envia el email usando el diccionario self.cleaned_data
        pass
```

views.py

```
from django.views.generic.edit import FormView
from biblioteca.forms import FormularioContactos

class VistaContactos(FormView):
    template_name = 'contactos.html'
    form_class = FormularioContactos
    success_url = '/gracias/'

    def form_valid(self, form):
    # Este método es llamado cuando el formulario valida los datos
    # A enviar. Debe devolver un HttpResponse
    form.send_email()
    return super(VistaContactos, self).form_valid(form)
```

contactos.html

```
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="enviar" value="Enviar mensaje" />
</form>
```

Vista de creación de objetos: *CreateView*

Clase: django.views.generic.edit.CreateView

Esta vista presenta un formulario que permite la creación de un objeto. Cuando se envían los datos del formulario, la vista se vuelve a mostrar si se produce algún error de validación (incluyendo, por supuesto, los mensajes pertinentes) o, en caso de que no se produzca ningún error de validación, guarda el objeto en la base de datos.

Ancestros (MRO)

Esta vista hereda los métodos y atributos de las siguientes vistas:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin

- django.views.generic.edit.BaseCreateView
- django.views.generic.edit.ModelFormMixin
- django.views.generic.edit.FormMixin
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

Atributos

- **template_name_suffix:** La página CreateView a mostrar, mediante una petición GET que usa como template_name_suffix a _form. Por ejemplo cambiando este atributo por _create_form para una vista para crear objetos, por ejemplo para el modelo Autor ocasionara que el valor predeterminado de template_name sea "biblioteca/autor_create_form.html".

- **object:** Cuando se usa CreateView se tiene acceso a self.object, el cual es el objeto creado. Si el objeto no ha sido creado, el valor será None.

Ejemplo:

Si quisiéramos permitir al usuario que creara nuevos autores en la base de datos, podríamos hacer algo como esto:

views.py

```
from django.views.generic.edit import CreateView
from biblioteca.models import Autor

class CrearAutor(CreateView):
    model = Autor
    fields = ['nombre, apellidos']
```

autor_form.html

```
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="enviar" value="Crear" />
</form>
```

Argumentos obligatorios

- **model:** El modelo Django del objeto a crear.

Observa que esta vista espera el *modelo* del objeto a crear, y no un QuerySet como el resto de las vistas que se han visto previamente.

Nombre de la plantilla

Si no se ha especificado ningún valor en template_name la vista usará como plantilla <app_label>/<model_name>_form.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla tendrá los siguientes valores:

- form: Una instancia de la clase ModelForm, que representa el formulario a utilizar. Esto te permite referirte de una forma sencilla a los campos del formulario desde la plantilla. Por ejemplo, si el modelo consta de dos atributos, nombre y dirección:

Vista para modificar objetos*: UpdateView*

Clase: django.views.generic.edit.UpdateView

Esta vista muestra un formulario para editar un objeto existente, vuelve a mostrar el formulario en caso de errores de validación (si los hay) y permite guardar los cambios en el objeto. Usa un formulario generado automáticamente por el modelo de la clase del objeto (A menos que se especifique manualmente una clase para el formulario).

Ancestros (MRO)

Esta vista hereda los métodos y atributos de las siguientes vistas:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseUpdateView
- django.views.generic.edit.ModelFormMixin
- django.views.generic.edit.FormMixin
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

Atributos

- *template_name_suffix:* La página *UpdateView* a mostrar, mediante una petición GET que usa como template_name_suffix a _form. Por ejemplo cambiando este atributo por _create_form para una vista para actualizar objetos, por ejemplo para el modelo Autor ocasionara que el valor predeterminado de template_name sea 'biblioteca/autor_create_form.html'.

- *object: Cuando* se usa UpdateView` se tiene acceso a ``self.object, el cual es el objeto modificado.

Ejemplo:

Siguiendo con el ejemplo, podemos proporcionar al usuario una interfaz de modificación de los datos de un autor con el siguiente código en el URLconf:

views.py

```
from django.views.generic.edit import UpdateView
from biblioteca.models import Autor

class ModificarAutor(UpdateView):
    model = Autor
    fields = ['nombre', 'apellidos']
    template_name_suffix = '_update_form'
```

autor_update_form.html

```
<form action="" method="post">{% csrf_token %}
  {{ form.as_p }}
   <input type="submit" value="Update" />
</form>
```

Argumentos obligatorios

- **model:** El modelo Django a editar. Hay que prestar atención a que es el modelo en sí, y no un objeto tipo QuerySet.

- **object_id:** El valor de la clave primaria del objeto a modificar. o bien un slug*:* El *slug* del objeto a modificar. Si se pasa este argumento, es obligatorio también el argumento ***slug_field.***

Argumentos opcionales

- ***slug_field:*** El nombre del campo en el que se almacena el valor del *slug* del sujeto. Es obligado usar este argumento si se ha indicado el argumento slug, pero no debe especificarse si hemos optado por identificar el objeto mediante su clave primaria, usando el argumento object_id.

Esta vista acepta los mismos argumentos opcionales que la vista de creación y, además, el argumento común template_object_name, explicado en la tabla C-1.

Nombre de la plantilla

Esta vista utiliza el mismo nombre de plantilla por defecto que la vista de creación (<app_label>/<model_name>_form.html).

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla tendrá los siguientes valores:

- **form***:* Una instancia de ModelForm que representa el formulario de edición del objeto.

- **object:** El objeto a editar (El nombre de esta variable puede ser diferente si se ha especificado el argumento template_object_name).

Vista de borrado de objetos: *DeleteView*

Clase: django.views.generic.edit.DeleteView

Una vista que muestra una página de confirmación y borrado de un objeto existente. Esta vista es muy similar a la dos anteriores: crear y modificar objetos. El propósito de esta vista es, sin embargo, permitir el borrado de objetos.

Si la vista es alimentada mediante GET, se mostrará una pantalla de confirmación (del tipo "¿Realmente quieres borrar este objeto?"). Si la vista se alimenta con POST, el objeto será borrado sin conformación.

Los argumentos son los mismos que los de la vista de modificación, así como las variables de contexto. El nombre de la plantilla por defecto para esta vista es <app_label>/<model_name>_confirm_delete.html.

Ancestros (MRO)

Esta vista hereda los métodos y atributos de las siguientes vistas:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseDeleteView
- django.views.generic.edit.DeletionMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View
- template_name_suffix

La pagina DeleteView a mostrar, mediante una peticion GET que usa como template_name_suffix a _confirm_delete. Por ejemplo cambiando este atributo por _check_delete para una vista para actualizar objetos, por ejemplo para el modelo Autor ocasionara que el valor predeterminado de template_name sea biblioteca/autor__check_delete.html.

Ejemplo:

Supongamos que queremos borrar un objeto Autor, esta es la forma en la que lo podemos hacer.

biblioteca/views.py

```
from django.views.generic.edit import DeleteView
from django.core.urlresolvers import reverse_lazy
from biblioteca.models import Author

class BorrarAutor(DeleteView):
    model = Autor
    success_url = reverse_lazy('lista-autores')
```

biblioteca/author_confirm_delete.html

```
<form action="" method="post">{% csrf_token %}
    <p> ¿Realmente quieres borrar este {{ object }}"?</p>
    <input type="enviar" value="Confirmar" />
</form>
```

Vistas genéricas basadas en fechas

Estas vistas genéricas basadas en fechas se suelen utilizar para organizar la parte de "archivo" de nuestro contenido. Los casos típicos son los archivos por año/mes/día de un periódico, o el archivo de una bitácora o *blog*.

En principio, estas vistas ignoran las fechas que estén situadas en el futuro.

Esto significa que si intentas visitar una página de un archivo que esté en el futuro, Django mostrará automáticamente un error 404 ("Página no encontrada"), incluso aunque hubiera objetos con esa fecha en el sistema.

Esto te permite publicar objetos por adelantado, que no se mostrarán públicamente hasta que se llegue a la fecha de publicación deseada.

Sin embargo, para otros tipos de objetos con fechas, este comportamiento no es el deseable (por ejemplo, un calendario de próximos eventos). Para estas vistas, podemos definir el argumento allow_future como True y de esa manera conseguir que los objetos con fechas futuras aparezcan (o permitir a los usuarios visitar páginas de archivo "en el futuro").

Índice de archivo: *ArchiveIndexView*

Clase: django.views.generic.dates.ArchiveIndexView

Esta vista proporciona un índice a nivel-superior donde se muestran los "últimos" objetos (es decir, los más recientes) según la fecha. Los objetos con fechas en el futuro no están incluidos, a menos que se establezca el atributo allow_future en True.

Ancestros (MRO)

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseArchiveIndexView
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Atributos

Además del contexto ofrecido por: django.views.generic.list.MultipleObjectMixin (via django.views.generic.dates.BaseDateListView), el contexto de las plantillas será:

- **date_list***:* Un objeto DateQuerySet que contiene todos las años según los cuales tengan objetos disponibles de acuerdo al queryset, representado como un objeto datetime.datetime en orden descendiente.

    1. Usa de forma predeterminada latest para el context_object_name.

    2. Usa de forma predeterminada _archive para el sufijo template_name_suffix.

3. Usa de forma predeterminada date_list por año, pero puede ser sobrescrito a mes o día usando el atributo date_list_period. Esto también se aplica a las vistas de las subclases.

Ejemplo:

Supongamos el típico editor que desea una página con la lista de sus últimos libros publicados. Suponiendo que tenemos un objeto Libro con un atributo tipo, fecha_publicacion, podemos usar la vista ArchiveIndexView para resolver este problema:

biblioteca/urls.py

```
from django.conf.urls import url
from django.views.generic.dates import ArchiveIndexView

from biblioteca.models import Libro

urlpatterns = [
    url(r'^ultimos-libros/$',   ArchiveIndexView.as_view(model=Libro,
        date_field="fecha_publicacion"), name="ultimos_libros"),
]
```

biblioteca/libro_archive.html

```
<ul>
    {% for libros in latest %}
        <li>{{ libros.fecha_publicacion }}: {{ libros.titulo }}</li>
    {% endfor %}
</ul>
```

Argumentos obligatorios

- **date_field:** El nombre de un campo tipo DateField o DateTimeField de los objetos que componen el QuerySet. La vista usará los valores de ese campo como referencia para obtener los últimos objetos.

- **queryset:** El QuerySet de objetos que forman el archivo o el model.

Argumentos opcionales

- **allow_future:** Un valor booleano que indica si los objetos "futuros" (es decir, con fecha de referencia en el futuro) deben aparecer o no.

Nombre de la plantilla

Si no se ha especificado template_name, se usará la plantilla <app_label>/<model_name>_archive.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto de la plantilla tendrá los siguientes valores:

- **date_list**: Una lista de objetos de tipo datetime.date que representarían todos los años en los que hay objetos, de acuerdo al queryset. Vienen ordenados de forma descendente, los años más recientes primero.

  Por ejemplo, para un blog que tuviera entradas desde el año 2003 hasta el 2006, la lista contendrá cuatro objetos de tipo datetime.date, uno para cada uno se esos años.

- **latest***:* Los últimos num_latest objetos en el sistema, considerándolos ordenados de forma descendiente por el campo date_field de referencia.

Archivos anuales: *YearArchiveView*

Clase: django.views.generic.dates.BaseYearArchiveView

Esta vista sirve para presentar archivos basados en años. Poseen una lista de los meses en los que hay algún objeto, y pueden mostrar opcionalmente todos los objetos publicados en un año determinado. Los objetos con fechas en el futuro no están incluidos, a menos que se establezca el atributo *allow_future* en True.

Ancestros (MRO)

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseYearArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Atributos

- ***make_object_list:*** Un valor booleano que especifica si debe recuperar la lista completa de objetos para este año y pasársela a la plantilla. Si es True la lista de objetos estará disponible en el contexto. Si es `False, el queryset usara el valor None como la lista de objetos. De forma predeterminada esta es False.

- ***get_make_object_list():*** Determina si un objeto de la lista, debe devolverse como parte de el contexto. Devuelve de forma predeterminada make_object_list

Atributos

Además del contexto ofrecido por: django.views.generic.list.MultipleObjectMixin (a través de django.views.generic.dates.BaseDateListView), el contexto de la plantilla contendrá:

- **date_list:** Un objeto DateQuerySet que contiene todos los meses en los que hay objetos disponibles, de acuerdo al queryset, representado como un objeto datetime.datetime en orden ascendente.

- **year:** Un objeto date que representa el año dado.

- **next_year**: Un objeto date que representa el primer día de el siguiente año, de acuerdo al allow_empty y allow_future.

- **previous_year:** Un objeto date que representa el primer día del año previo, de acuerdo al allow_empty y allow_future.

  Usa de forma predeterminada _archive_year como el nombre del sufijo de plantilla para template_name_suffix.

Ejemplo:

Como ejemplo, vamos a ampliar el ejemplo anterior incluyendo una vista que muestre todos los libros publicados en un determinado año:

biblioteca/views.py

```
from django.views.generic.dates import YearArchiveView
from biblioteca.models import Libro

class LibrosAnuales(YearArchiveView):
    queryset = Libro.objects.all()
    date_field = "fecha_publicacion"
    make_object_list = True
    allow_future = True
```

biblioteca/urls.py

```
from django.conf.urls import url
from biblioteca.views import LibrosAnuales

urlpatterns = [
    url(r'^(?P<year>[0-9]{4})/$', LibrosAnuales.as_view(),
        name="libros_anuales"),
]
```

biblioteca/libros_archive_year.html

```
<ul>
    {% for fecha in date_list %}
      <li>{{ fecha|date }}</li>
    {% endfor %}
</ul>
<div>
    <h1>Todos los libros del {{ year|date:"Y" }}</h1>
{% for libros in object_list %}
<p>
    {{ libros.titulo }} - {{ libros.fecha_publicacion|date:"F j, Y" }}
</p>
{% endfor %}
</div>
```

Argumentos obligatorios

- **date_field:** Igual que en ArchiveIndexView (Véase la sección previa).
- **queryset:** El QuerySet de objetos archivados.
- **year:** El año, con cuatro dígitos, que la vista usará para mostrar el archivo (Como se ve en el ejemplo, normalmente se obtiene de un parámetro en la URL).

Argumentos opcionales

- **make_object_list:** Un valor booleano que indica si se debe obtener la lista completa de objetos para este año y pasársela a la plantilla. Si es True, la lista de objetos estará disponible para la plantilla con el nombre de object_list (Aunque este nombre podría ser diferente; véase la información sobre object_list en la siguiente explicación sobre "**Atributos** de plantilla"). Su valor por defecto es False.
- **allow_future:** Un valor booleano que indica si deben incluirse o no en esta vista las fechas "en el futuro".

Nombre de la plantilla

Si no se especifica ningún valor en name, la vista usará la plantilla <app_label>/<model_name>_archive_year.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto de la plantilla, tendrá los siguientes valores:

- **date_list:** Una lista de objetos de tipo datetime.date, que representan todos los meses en los que hay disponibles objetos en un año determinado, de acuerdo al contenido del queryset, en orden ascendente.
- **year:** El año a mostrar, en forma de cadena de texto con cuatro dígitos.
- **object_list:** Si el parámetro make_object_list es True, esta variable será una lista de objetos cuya fecha de referencia cae en en año a mostrar, ordenados por fecha. El nombre de la variable depende del parámetro template_object_name, que es 'object' por defecto. Si template_object_name fuera 'foo', el nombre de esta variable sería foo_list.

  Si make_object_list es False, object_list será una lista vacía.

Archivos mensuales: *MonthArchiveView*

Clase: django.views.generic.dates.BaseMonthArchiveView

Esta vista proporciona una representación basada en meses, en la que se muestran todos los objetos cuya fecha de referencia caiga en un determinado mes y año. Los

objetos con fechas en el futuro no están incluidos, a menos que se establezca el atributo allow_future en True.

Ancestros (MRO)

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseMonthArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.MonthMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Atributos

Además del contexto ofrecido por: MultipleObjectMixin (a través de BaseDateListView), el contexto de la plantilla contendrá:

- **date_list:** Un objeto DateQuerySet que contiene todos los días que contienen objetos disponibles en el mes dado, de acuerdo al queryset representado por el objeto datetime.datetime en orden ascendente.

- **month:** Una objeto de la clase date que representa en mes dado.

- **next_month**: Un objeto de la clase date que representa el primer día de el siguiente mes, de acuerdo al atributo allow_empty y allow_future.

- **previous_month**: Un objeto de la clase date que representa el primer día del mes anterior, de acuerdo al atributo allow_empty y allow_future.

Usa de forma predeterminada _archive_month como el nombre del sufijo de plantilla para template_name_suffix.

Ejemplo

Siguiendo con nuestro ejemplo, añadir una vista mensual a nuestra aplicación, debería ser algo sencillo:

biblioteca/views.py

```
from django.views.generic.dates import MonthArchiveView
from biblioteca.models import Libro

class LibrosPorMes(MonthArchiveView):
    queryset = Libro.objects.all()
    date_field = "fecha_publicacion"
    make_object_list = True
    allow_future = True
```

biblioteca/urls.py

```
from django.conf.urls import url
from biblioteca.views import LibrosPorMes

urlpatterns = [
    # Ejemplo: /2012/agosto/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/$', LibrosPorMes.as_view(),
        name="libros_mes"),
    # Ejemplo: /2012/08/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[0-9]+)/$',
        LibrosPorMes.as_view(month_format='%m'), name="libros_mes_numerico"),
]
```

libro_archive_month.html

```
<ul>
    {% for libro in object_list %}
        <li>{{ libro.fecha_publicacion|date:"F j, Y" }}: {{ libro.titulo }}</li>
    {% endfor %}
</ul>
<p>
    {% if previous_month %}
        Mes anterior: {{ previous_month|date:"F Y" }}
    {% endif %}

    {% if next_month %}
        Mes siguiente: {{ next_month|date:"F Y" }}
    {% endif %}
</p>
```

Argumentos obligatorios

- **year:** El año a mostrar, en forma de cadena de texto con cuatro dígitos.
- **month**: El mes a mostrar, formateado de acuerdo con el argumento month_format.
- **queryset:** El QuerySet de objetos archivados.
- **date_field**: El nombre del campo de tipo DateField o DateTimeField en el modelo usado para el QuerySet que se usará como fecha de referencia.

Argumentos opcionales

- **month_format:** Una cadena de texto que determina el formato que debe usar el parámetro month. La sintaxis a usar debe coincidir con la de la función time.strftime (La documentación de esta función se puede consultar en http://www.djangoproject.com/r/python/strftime/). Su valor por defecto es "%b", que significa el nombre del mes, en inglés, y abreviado a tres letras (Es decir, "jan", "feb", etc.). Para cambiarlo de forma que se usen números, hay que utilizar como cadena de formato "%m".

- **allow_future***:* Un valor booleano que indica si deben incluirse o no en esta vista las fechas "en el futuro", igual al que hemos visto en otras vistas anteriores.

Nombre de la plantilla

Si no se especifica ningún valor en template_name, la vista usará como plantilla <app_label>/<model_name>_archive_month.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla contendrá los siguientes valores:

- **month**: Un objeto de tipo datetime.date que representa el mes y año de referencia.

- **next_month:** Un objeto de tipo datetime.date que representa el primer día del siguiente mes. Si el siguiente mes cae en el futuro, valdrá None.

- **previous_month:** Un objeto de tipo datetime.date que representa el primer día del mes anterior. Al contrario que next_month, su valor nunca será None.

- **object_list**: Una lista de objetos cuya fecha de referencia cae en en año y mes a mostrar. El nombre de la variable depende del parámetro template_object_name, que es 'object' por defecto.

  Si *template_object_name* fuera 'foo', el nombre de esta variable sería foo_list.

Archivos semanales*: WeekArchiveView*

Clase: django.views.generic.dates.BaseWeekArchiveView

Esta vista muestra todos los objetos de una semana determinada. Los objetos con fechas en el futuro no están incluidos, a menos que se establezca el atributo allow_future en True.

Ancestros (MRO)

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseWeekArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.WeekMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Atributos

Además del contexto ofrecido por: MultipleObjectMixin (a través de BaseDateListView), el contexto de la plantilla contendrá:

- **week:** Una objeto date que representa el primer día de la semana dada.
- **next_week:** Un objeto date que representa el primer día de la siguiente semana, de acuerdo al atributo allow_empty y allow_future.
- **previous_week:** Un objeto date que representa el primer día de la semana previa, de acuerdo a allow_empty y allow_future.

  Usa de forma predeterminada _archive_week como el nombre del sufijo de plantilla para template_name_suffix.

Nota: Por consistencia con las Librerías de manejo de fechas de Python, Django asume que el primer día de la semana es el domingo.

Ejemplo:

Siguiendo con nuestro ejemplo, añadir una vista semanal a nuestra aplicación, no debería ser muy complicado.

biblioteca/views.py.html

```
from django.views.generic.dates import WeekArchiveView
from biblioteca.models import Libro

class LibrosSemanales(WeekArchiveView):
    queryset = Libro.objects.all()
    date_field = "fecha_publicacion"
    make_object_list = True
    week_format = "%W"
    allow_future = True
```

biblioteca/urls.py

```
from django.conf.urls import url
from biblioteca.views import LibrosSemanales
urlpatterns = [
# Example: /2012/week/23/
url(r'^(?P<year>[0-9]{4})/week/(?P<week>[0-9]+)/$',
LibrosSemanales.as_view(),
name="libros-semanales"),
]
```

article_archive_week.html

```
<h1>Semana {{ week|date:'W' }}</h1>
<ul>
    {% for libros in object_list %}
        <li>{{ libros.fecha_publicacion|date:"F j, Y" }}: {{ libro.titulo }}</li>
    {% endfor %}
```

```
</ul>

<p>
   {% if previous_week %}
      Semana anterior: {{ previous_week|date:"F Y" }}
   {% endif %}

   {% if previous_week and next_week %}--{% endif %}
      {% if next_week %}
         Semana siguiente: {{ next_week|date:"F Y" }}
   {% endif %}
</p>
```

En este ejemplo, mostramos la salida de el número de semanas. El valor predeterminado para week_format en la vista WeekArchiveView usa '%U' el cual está basado en el sistema de semanas manejado en los Estados Unidos, cuyo inicio de semana es el domingo. El formato ISO usa el formato de semanas '%W', en este formato la semana comienza el lunes. El formato '%W' es el mismo en ambas funciones: strftime() y en el filtro date.

Sin embargo el filtro date del el sistema de plantillas no tiene un equivalente, para la salida en el formato que soporta el sistema de semanas US. El filtro date '%U', muestra por salida el numero de segundos desde la época Unix.

Argumentos obligatorios

- **year:** El año, con cuatro dígitos (Una cadena de texto).
- **week:** La semana del año (Una cadena de texto).
- **queryset:** El QuerySet de los objetos archivados.
- **date_field*:*** El nombre del campo de tipo DateField o DateTimeField en el modelo usado para el QuerySet que se usará como fecha de referencia.

Argumentos opcionales

- ***allow_future:*** Un valor booleano que indica si deben incluirse o no en esta vista las fechas “en el futuro”.

Nombre de la plantilla

Si no se ha especificado ningún valor en template_name la vista usará como plantilla <app_label>/<model_name>_archive_week.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla contendrá los siguientes valores:

- **week:** Un objeto de tipo datetime.date, cuyo valor es el primer día de la semana considerada.

- **object_list:** Una lista de objetos disponibles para la semana en cuestión. El nombre de esta variable depende del parámetro template_object_name, que es 'object' por defecto. Si template_object_name fuera 'foo', el nombre de esta variable sería foo_list.

Archivos diarios*: DayArchiveView*

Clase: django.views.generic.dates.BaseDayArchiveView

Esta vista muestra todos los objetos para un día determinado. Los objetos con fechas en el futuro muestran un error 404, no importa si existen objetos a menos que se establezca el atributo allow_future en True.

Ancestros (MRO)

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseDayArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.MonthMixin
- django.views.generic.dates.DayMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Atributos

Además del contexto ofrecido por: MultipleObjectMixin (a través de BaseDateListView), el contexto de la plantilla contendrá:

- ***day:*** Un objeto date que representa el día dado.

- ***next_day:*** Un objeto date object que representa el día siguiente, de acuerdo al allow_empty y allow_future.

- ***previous_day:*** Un objeto date que representa el día anterior, de acuerdo al atributo allow_empty y allow_future.

- ***next_month:*** Un objeto de la clase date que representa el primer día de el siguiente mes, de acuerdo al atributo allow_empty y allow_future.

- ***previous_month:*** Un objeto de la clase date que representa el primer día del mes anterior, de acuerdo al atributo allow_empty y allow_future.

  Usa de forma predeterminada _archive_day como el nombre del sufijo de plantilla para template_name_suffix.

Ejemplo:

Siguiendo con nuestro ejemplo, añadir una vista diaria de objetos a nuestra aplicación, no debería ser más complicada que las anteriores.

biblioteca/views.py

```
from django.views.generic.dates import DayArchiveView
from biblioteca.models import Libro

class LibrosDiarios(DayArchiveView):
    queryset = Libro.objects.all()
    date_field = "fecha_publicacion"
    make_object_list = True
    allow_future = True
```

biblioteca/biblioteca/urls.py

```
from django.conf.urls import url
from biblioteca.views import LibrosDiarios

urlpatterns = [
# Ejemplo: /2012/nov/10/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/(?P<day>[0-9]+)/$',
        LibrosDiarios.as_view(), name="libros-diarios"),
]
```

biblioteca/libro_archive_day.html

```
<h1>{{ day }}</h1>
<ul>
    {% for libros in object_list %}
        <li>{{ libros.fecha_publicacion|date:"F j, Y" }}: {{ libro.titulo }}</li>
    {% endfor %}
</ul>
<p>

    {% if previous_day %}
        Dia anterior: {{ previous_day }}
    {% endif %}
        {% if previous_day and next_day %}--{% endif %}
            {% if next_day %}
                Siguiente dia: {{ next_day }}
        {% endif %}
</p>
```

Argumentos obligatorios

- **year**: El año, con cuatro dígitos (Una cadena de texto).
- **month**: El mes, formateado de acuerdo a lo indicado por el argumento month_format.
- **day:** El día, formateado de acuerdo al argumento day_format.

- **queryset**: El QuerySet de los objetos archivados.

- **date_field***:* El nombre del campo de tipo DateField o DateTimeField en el modelo usado para el QuerySet que se usará como fecha de referencia.

Argumentos opcionales

- **month_format:** Una cadena de texto que determina el formato que debe usar el parámetro month. Hay una explicación más detallada en la sección de "Archivos mensuales", incluida anteriormente.

- **day_format:** Equivalente a month_format, pero para el día. Su valor por defecto es "%d" (que es el día del mes como número decimal y relleno con ceros de ser necesario; 01-31).

- **allow_future:** Un valor booleano que indica si deben incluirse o no en esta vista las fechas "en el futuro".

Nombre de la plantilla

Si no se ha especificado ningún valor en template_name la vista usará como plantilla <app_label>/<model_name>_archive_day.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla tendrá los siguientes valores:

- **day:** Un objeto de tipo datetime.date cuyo valor es el del día en cuestión.

- **next_day**: Un objeto de tipo datetime.date que representa el siguiente día. Si cae en el futuro, valdrá None.

- **previous_day**: Un objeto de tipo datetime.date que representa el día anterior. Al contrario que next_day, su valor nunca será None.

- **object_list**: Una lista de objetos disponibles para el día en cuestión. El nombre de esta variable depende del parámetro template_object_name, que es object por defecto. Si template_object_name fuera foo, el nombre de esta variable sería foo_list.

Archivo para hoy*: TodayArchiveView*

Clase: django.views.generic.dates.BaseTodayArchiveView

Esta vista muestra todos los objetos cuya fecha de referencia sea *hoy*. Es parecida a django.views.generic.dates.DayArchiveView, excepto que no se utilizan los argumentos *year/month/day*, ya que esos datos se obtendrán de la fecha actual.

Ancestros (MRO)

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin

- django.views.generic.dates.BaseTodayArchiveView
- django.views.generic.dates.BaseDayArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.MonthMixin
- django.views.generic.dates.DayMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Usa de forma predeterminada _archive_today como el nombre del sufijo de plantilla para template_name_suffix.

Ejemplo:

Siguiendo con ejemplo anterior, podemos añadir una vista para mostrar los objetos del día de hoy de la siguiente forma.

biblioteca/views.py

```
from django.views.generic.dates import TodayArchiveView
from biblioteca.models import Libro

class LibrosPublicadosHoy(TodayArchiveView):
    queryset = Libro.objects.all()
    date_field = "fecha_publicacion"
    make_object_list = True
    allow_future = True
```

biblioteca/urls.py

```
from django.conf.urls import url
from myapp.views import LibrosPublicadosHoy

urlpatterns = [
    url(r'^hoy/$', LibrosPublicadosHoy.as_view(), name="libros-publicados-hoy"),
]
```

¿Donde está la plantilla para todayarchiveview?

Esta vista usa de forma predeterminada la misma plantilla que la clase DayArchiveView, como en el ejemplo anterior. Si necesitas una plantilla diferente, establece un atributo template_name para utilizar el nombre de la nueva plantilla.

Páginas de detalle basadas en fecha: *DateDetailView*

Clase: django.views.generic.dates.BaseDateDetailView

Esta vista se usa para representar un objeto individual. Los objetos con fechas en el futuro muestran un error 404, no importa si existen los objetos a menos que se establezca el atributo allow_future en True.

Ancestros (MRO)

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseDateDetailView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.MonthMixin
- django.views.generic.dates.DayMixin
- django.views.generic.dates.DateMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

Atributos

Incluye el único objeto asociado al modelo especificado en DateDetailView.

Usa de forma predeterminada _detail como el nombre del sufijo de plantilla para template_name_suffix.

Esta vista tiene una URL distinta de la vista DetailView; mientras que la última usa una URL como, por ejemplo, /entradas/<slug>/, esta usa una URL en la forma /entradas/2006/aug/27/<slug>/.

Nota: Si estás usando páginas de detalle basadas en fechas con *slugs* en la URL, lo más probable es que quieras usar la opción unique_for_date en el campo *slug*, de forma que se garantice que los *slugs* nunca se duplican para una misma fecha.

Ejemplo:

Esta vista tiene una (pequeña) diferencia con las demás vistas basadas en fechas que hemos visto anteriormente, y es que necesita que le especifiquemos de forma inequívoca el objeto en cuestión; esto lo podemos hacer con el identificador del objeto *pk* o con un campo de tipo *slug*.

Como el objeto que estamos usando en el ejemplo no tiene ningún campo de tipo *slug*, usaremos el identificador para la URL. Normalmente se considera una buena práctica usar un campo *slug*, pero no lo haremos en aras de simplificar el ejemplo.

biblioteca/urls.py

```
from django.conf.urls import url
from django.views.generic.dates import DateDetailView

from biblioteca.models import Libro
urlpatterns = [
    url(r'^(?P<year>[0-9]+)/(?P<month>[-\w]+)/(?P<day>[0-9]+)/(?P<pk>[0-9]+)/$',
        DateDetailView.as_view(model=Libro, date_field="fecha_publicacion"),
        name="libros-detalle-por-fecha"),
]
```

biblioteca/libro_detail.html

```
<h1>{{ object.titulo }}</h1>
```

Argumentos obligatorios

- **year:** El año, con cuatro dígitos (Una cadena de texto).
- **month:** El mes, formateado de acuerdo a lo indicado por el argumento month_format
- **day:** El día, formateado de acuerdo al argumento day_format.
- **queryset**: El QuerySet que contiene el objeto.
- **date_field:** El nombre del campo de tipo DateField o DateTimeField en el modelo usado para el QuerySet que se usará como fecha de referencia.

Y también habrá que especificar, o bien un:

- **object_id:** El valor de la clave primaria del objeto, o bien un: ***slug:*** El *slug* del objeto. Si se utiliza este argumento, es obligatorio especificar un valor para el argumento slug_field (que describiremos en la siguiente sección).

Argumentos opcionales

- **allow_future:** Un valor booleano que indica si deben incluirse o no en esta vista las fechas "en el futuro".
- **day_format:** Equivalente a month_format, pero para el día. Su valor por defecto es "%d" (que es el día del mes como número decimal y relleno con ceros de ser necesario; 01-31).
- **month_format:** Una cadena de texto que determina el formato que debe usar el parámetro month. Hay una explicación más detallada en la sección de "Archivos mensuales", incluida anteriormente.
- **slug_field:** El nombre del atributo que almacena el valor del slug*. Es obligatorio incluirlo si se ha usado el argumento slug, y no debe aparecer si se ha especificado el argumento object_id.
- **template_name_*field***: El nombre de un atributo del objeto cuyo valor se usará como el nombre de la plantilla a utilizar. De esta forma, puedes almacenar en tu objeto la plantilla a usar.

Nombre de la plantilla

Si no se ha especificado ningún valor en template_name la vista usará como plantilla <app_label>/<model_name>_detail.html.

Atributos de la plantilla

Además de los valores que se puedan haber definido en el contexto, la plantilla contendrá los siguientes valores:

- **object***:* El objeto en sí mismo, el nombre de esta variable depende del parámetro template_object_name, que es object por defecto. Si template_object_name fuera foo, el nombre de esta variable sería foo.

Todas las vistas genéricas basadas en clases listadas anteriormente, corresponden y heredan de la vista Base, únicamente difieren de ella en que no incluyen la clase MultipleObjectTemplateResponseMixin (para las vistas de archivos) o SingleObjectTemplateResponseMixin (para la clase DateDetailView).

APÉNDICE D

Variables de configuración

El archivo de configuración *setting.py* contiene toda la configuración de tu instalación Django. Este apéndice explica cómo funcionan la mayoría de las variables de configuración y qué variables de configuración están disponibles.

Nota: A medida que Django crece, es ocasionalmente necesario agregar, quitar o cambiar algunas variables de configuración. Debes siempre buscar la información más reciente en la documentación oficial en línea, que se encuentra disponible en el sitio del proyecto http://www.djangoproject.com/documentation/.

Qué es un archivo de configuración

Un *archivo de configuración* es sólo un módulo Python con variables a nivel de módulo.

Un par de ejemplos de variables de configuración:

```
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
```

Debido a que un archivo de configuración es un módulo Python, las siguientes afirmaciones son ciertas:

- Debe ser código Python válido; no se permiten los errores de sintaxis.
- El mismo puede asignar valores a las variables dinámicamente usando sintaxis normal de Python, por ejemplo:

  ```
  MI_CONFIGURACION = [str(i) for i in range(30)]
  ```

- El mismo puede importar valores desde otros archivos de configuración.

Valores predeterminados

No es necesario que un archivo de configuración de Django defina una variable de configuración si es que no es necesario. Cada variable de configuración tiene un valor por omisión sensato. Dichos valores por omisión residen en el archivo django/conf/global_settings.py.

Este es el algoritmo que usa Django cuando compila los valores de configuración:

1. Carga las variables de configuración desde *global_settings*.

2. Carga las variables de configuración desde el archivo de configuración especificado, reemplazando de ser necesario los valores globales previos.

Nota que un archivo de configuración *no* debe importar desde global_settings, ya que eso sería redundante.

¿Cómo saber cuáles variables de configuración has cambiado?

Existe una manera fácil de ver cuáles de tus variables de configuración difieren del valor por omisión. El comando manage.py diffsettings visualiza las diferencias entre el archivo de configuración actual y los valores por omisión de Django.

manage.py es descrito con más detalle en el Apéndice G.

Usando variables de configuración en código Python

En tus aplicaciones Django, usa variables de configuración importando el objeto *django.conf.settings*, por ejemplo:

```
from django.conf import settings

if settings.DEBUG:
    # Haz algo
```

Nota que django.conf.settings no es un módulo – es un objeto. De manera que no es posible importar variables de configuración individualmente.

```
from django.conf.settings import DEBUG # No trabaja.
```

Ten en cuenta también que tu código *no* debe importar ni desde global_settings ni desde tu propio archivo de configuración. django.conf.settings provee abstracción para los conceptos de variables de configuración por omisión y variables de configuración específicas de un sitio; presenta una única interfaz. También desacopla el código que usa variables de configuración de la ubicación de dicha configuración.

Modificando variables de configuración en tiempo de ejecución

No debes alterar variables de configuración en tiempo de ejecución. Por ejemplo, no hagas esto en una vista:

```
from django.conf import settings

settings.DEBUG = True # ¡No hagas esto!
```

El único lugar en el que debes asignar valores a settings es en un archivo de configuración.

Seguridad

Debido que un archivo de configuración contiene información importante, tal como la contraseña de la base de datos, debes hacer lo que esté e tus manos para limitar el acceso al mismo. Por ejemplo, cambia los permisos de acceso en el sistema de

archivos de manera que solo tú y el usuario de tu servidor Web puedan leerlo. Esto es especialmente importante en un entorno de alojamiento compartido.

Creando tus propias variables de configuración: DJANGO_SETTINGS_MODULE

No existe nada que impida que crees tus propias variables de configuración, para tus propias aplicaciones Django. Sólo sigue las siguientes convenciones:

Usa nombres de variables en mayúsculas.

Para configuraciones que sean secuencias, usa tuplas en lugar de listas. Las variables de configuración deben ser consideradas inmutables y no deben ser modificadas una vez que se las ha definido. El usar tuplas refleja esa semántica.

No reinventes una variable de configuración que ya existe.

Indicando la configuración: DJANGO_SETTINGS_MODULE

Cuando usas Django tienes que indicarle qué configuración estás usando. Haz esto mediante el uso de de la variable de entorno DJANGO_SETTINGS_MODULE.

El valor de DJANGO_SETTINGS_MODULE debe respetar la sintaxis de rutas de Python (por ej. mysite.settings. Notar que el módulo de configuración debe ser encontrarse en la ruta de búsqueda para las importaciones de Python (PYTHONPATH).

■**Nota:** Puedes encontrar una buena guía acerca de *PYTHONPATH* en http://diveintopython.org/getting_to_know_python/everything_is_an_object.html.

La utilidad django-admin.py

Cuando usas django-admin.py (ver Apéndice F), puedes ya sea fijar el valor de la variable de entorno una vez o especificar explícitamente el módulo de configuración cada vez que ejecutes la utilidad.

Este es un ejemplo usando el shell Bash de Unix:

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin.py runserver
```

Este es otro ejemplo, esta vez usando el shell de Windows:

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin.py runserver
```

Usa el argumento de línea de comandos *--settings* para especificar el módulo de configuración en forma manual:

```
django-admin.py runserver --settings=mysite.settings
```

La utilidad manage.py creada por el comando startproject como parte del esqueleto del proyecto asigna un valor a DJANGO_SETTINGS_MODULE en forma automática; revisa el Apéndice G si deseas conocer más sobre manage.py.

En el servidor (mod_wsgi)

En tu entorno del servidor activo, necesitarás indicarle a WSGI application qué archivo de configuración debe usar. Haz eso con *os.environ*:

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'misitio.settings'
```

Usando variables de configuración sin usar DJANGO_SETTINGS_MODULE

Es algunos casos, querrás saltearte la variable de entorno DJANGO_SETTINGS_MODULE.

Por ejemplo, si estás usando el sistema de plantillas en forma aislada, muy probablemente no desearás tener que preparar una variable de entorno que apunte a un módulo de configuración.

En esos casos, puedes fijar los valores de las variables de configuración de Django manualmente. Haz esto llamando a django.conf.settings.configure(). Este es un ejemplo:

```
from django.conf import settings

settings.configure(DEBUG=True, TEMPLATE_DEBUG=True,
TEMPLATE_DIRS=('/home/web-apps/myapp', '/home/web-apps/base'))
```

Pásale a configure() tantos argumentos de palabra clave como desees, con cada argumento representando una variable de configuración y su valor. Cada nombre de argumento debe estar escrito totalmente en mayúsculas, con el mismo nombre que la variable de configuración que ya se describieron. Si una variable de configuración no es pasada a configure() y es necesario luego, Django usará el valor por omisión respectivo.

El configurar Django de esta manera es en general necesario – y, en efecto, recomendado, cuando usas una parte del framework dentro de una aplicación más grande.

En consecuencia, cuando es configurado vía settings.configured(), Django no hará modificación alguna a las variables de entorno del proceso (revisa la explicación acerca de TIME_ZONE más adelante en este apéndice para conocer por qué habría de ocurrir esto). Asumimos que en esos casos ya tienes completo control de tu entorno.

Variables de configuración personalizados

Si te gustaría que los valores por omisión provinieran desde otra ubicación diferente a django.conf.global_settings, puedes pasarle un módulo o una clase que provea las variables de configuración por omisión como el argumento default_settings (o como el primer argumento posicional) en la llamada a configure().

En este ejemplo, las variables de configuración por omisión se toman desde myapp-defaults, y se fija el valor de DEBUG en True, independientemente de su valor en myapp_defaults:

```
from django.conf import settings
from myapp import myapp_defaults
```

```
settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

El siguiente ejemplo, que usa myapp_defaults como un argumento posicional, es equivalente:

```
settings.configure(myapp_defaults, DEBUG = True)
```

Normalmente, no necesitarás sobrescribir los valores por omisión de esta manera. Los valores por omisión provistos por Django son suficientemente sensatos como para que puedas usarlos. Ten en cuenta que si pasas un nuevo valor por omisión, este *reemplaza* completamente los valores de Django, así que debes especificar un valor para cada variable de configuración posible que pudiera ser usado en el código que estás importando. Examina django.conf.settings.global_settings para ver la lista completa.

Usa configure() o DJANGO_SETTINGS_MODULE

Si no estás fijando la variable de entorno DJANGO_SETTINGS_MODULE, debes llamar a configure() en algún punto antes de usar cualquier código que lea las variables de configuración.
Si no fijas DJANGO_SETTINGS_MODULE y no llamas a configure(), Django lanzará una excepción EnvironmentError la primera vez que se accede a una variable de configuración.

Si fijas el valor de DJANGO_SETTINGS_MODULE, luego accedes a los valores de las variables de configuración de alguna manera, y *entonces* llamas a configure(), Django lanzará un EnvironmentError indicando que la configuración ya ha sido preparada.

También es un error el llamar a configure() más de una vez, o llamar a configure luego de que ya se ha accedido a alguna variable de configuración.

En resumen: Usa exactamente una vez ya sea configure() o DJANGO_SETTINGS_MODULE. No ambos, y no ninguno.

Variables de configuración disponibles

Las siguientes secciones consisten de una lista completa de todas las variables de configuración en orden alfabético, y sus valores por omisión.

■ **Nota:** Ten cuidado al sobrescribir alguna configuración, especialmente cuando el valor predeterminado no está vacío, es un diccionario o una tupla, tal como MIDDLEWARE_CLASSES y TEMPLATE_CONTEXT_PROCESSORS. Asegúrate que los componentes requeridos estén disponibles para usar esta característica de Django.

ABSOLUTE_URL_OVERRIDES

Valor por omisión: {} (Diccionario vacío)

Un diccionario enlazando cadenas app_label.model_name a funciones que toman un objeto modelo y retornan su URL. Esta es una forma de sobrescribir métodos get_absolute_url() en cada instalación. Un ejemplo:

```
ABSOLUTE_URL_OVERRIDES = {
    'blogs.weblog': lambda o: "/blogs/%s/" % o.slug,
    'news.story': lambda o: "/stories/%s/%s/" % (o.pub_year, o.slug),
}
```

Nota que el nombre del modelo usado en esta variable de configuración debe estar escrito totalmente en mayúsculas, con independencia de la combinación de mayúsculas y minúsculas del nombre real de la clase del modelo.

ABSOLUTE_URL_OVERRIDES no funciona en modelos que no tienen declarado un metodo get_absolute_url().

ADMINS

Valor por omisión: () (Tupla vacía)

Una tupla que enumera las personas que recibirán notificaciones de errores en el código. Cuando DEBUG=False y una vista laza una excepción, Django enviará a esta gente un e-mail con la información completa de la información. Cada miembro de la tupla debe ser una tupla de (Nombre completo, dirección de e-mail), por ejemplo:

```
(('John', 'john@example.com'), ('Mary', 'mary@example.com'))
```

Observa que Django el enviará e-mail a *todas* estas personas cada vez que ocurra un error.

ALLOWED_HOSTS

Valor por omisión: [] (Lista vacía)

Una lista de cadenas que representa el nombre del host/dominio que usa el sitio de Django. Se trata de una medida de seguridad, que impide que un atacante puede envenenar la cache y resetear contraseñas enviando emails con links a sitios maliciosos, enviando peticiones HTTP con cabeceras falsas Host, lo cual es posible incluso bajo muchas configuraciones aparentemente-seguras del servidor web.

ALLOWED_INCLUDE_ROOTS

***Valor por omisión*: ()** (Tupla vacía)

Una tupla de cadenas que representan prefijos permitidos para la etiqueta de plantillas {% ssi %}. Se trata de una medida de seguridad, que impide que los autores de plantillas puedan acceder a archivos a los que no deberían acceder.

Por ejemplo, si ALLOWED_INCLUDE_ROOTS es ('/home/html', '/var/www') entonces {% ssi /home/html/foo.txt %} funcionaría pero {% ssi /etc/passwd %} no.

APPEND_SLASH

Valor por omisión: True

Esta variable de configuración indica si debe anexarse barras al final de las URLs. Se usa solamente si está instalado el CommonMiddleware (ver *capítulo 17*).

CACHES

Valor por omisión:

```
CACHES{
   'default': {
      'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
   }
}
```

Un diccionario que contiene la configuración para todas las caches que se usaran con Django. Es un diccionario jerarquizado cuyos contenidos asocian en forma de alias un diccionario que contiene las opciones para usar la cache de forma individual.

La configuración de CACHES debe configurar el valor predeterminado default cache; y cualquier numero adicional de caches debe ser especificado. Si estas usando algún tipo de backend u otra cache en memoria o necesitas definir múltiples caches, necesitas definir otras opciones. Las siguientes opciones de cache están disponibles:

BACKEND

***Valor por omisión*:** '' (Una cadena vacía)

El backend para usar como cache. Los backends incorporados en la cache son:

```
'django.core.cache.backends.db.DatabaseCache'
'django.core.cache.backends.dummy.DummyCache'
'django.core.cache.backends.filebased.FileBasedCache'
'django.core.cache.backends.locmem.LocMemCache'
'django.core.cache.backends.memcached.MemcachedCache'
'django.core.cache.backends.memcached.PyLibMCCache'
```

Puedes usar algún otro tipo de almacenamiento para la cache o backend configurando BACKEND con la ruta completa a la clase backend que estés usando. Por ejemplo mipaquete.backends.whatever.WhateverCache

KEY_FUNCTION

Una cadena que contiene la ruta a la función (o cualquier llamable) que define la forma en que se compone el prefijo, version y key en la clave de la cache final. El valor predeterminado es equivalente a la función:

```
def make_key(key, key_prefix, version):
   return ':'.join([key_prefix, str(version), key])
```

Puedes usar cualquier función clave que quieras, siempre que tenga los mismos argumentos.

KEY_PREFIX

Valor por omisión: '' (Cadena Vacía)

Una cadena que estará automáticamente incluida (agregada por omisión) en todas la claves de la cache usadas por el servidor.

LOCATION

Valor por omisión: '' (Cadena Vacía)

La localización de la cache a utilizar. Ésta puede ser el directorio para usar un archivo como sistema de cache, un host o un puerto para el servidor de memcache, o simplemente un nombre para identificar la memoria local que se esté usando, por ejemplo:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

OPTIONS

Valor por omisión: None

Parámetros extra para pasar a la cache. Los parámetros disponibles varían dependiendo del tipo de backend usado para la cache.

TIMEOUT

Valor por omisión: 300

El número de segundos antes de que una entrada en la cache expire. Si el valor de la configuración es None, la entrada en la cache no expira.

VERSION

Valor por omisión: 1

El valor predeterminado para el número de versión, generado por el servidor.

CACHE_MIDDLEWARE_ALIAS

Valor por omisión: default

La conexión a la cache a usar por el middleware de cache.

CACHE_MIDDLEWARE_KEY_PREFIX

Valor por omisión: '' (Cadena vacía)

El prefijo de las claves de cache que debe usar el middleware de cache (ver *capítulo 17*).

CACHE_MIDDLEWARE_SECONDS

Valor por omisión: 600

El valor predeterminado para el numero de segundos que se mantendrá en cache una página, cuando se usen el middleware de cacheo o el decorador cache_page().

CSRF_COOKIE_AGE

Valor por omisión: 31449600 (1 año, en segundos)

La edad de las cookies CSRF, en segundos.

La razón para configurar el tiempo de vida y expiración de las cookies, es para evitar problemas en el caso de que se cierre el navegador de un usuario o una página de marcadores y se cargue la página desde la cache del navegador. Sin cookies persistentes, los formularios para subir datos fallaran.

Algunos navegadores (especialmente Internet Explorer) pueden rechazar el uso de cookies persistentes o pueden tener índices de cookies corrompidos en el disco, por consiguiente causan que la comprobación de protección CSRF falle (a veces intermitentemente). Cambia esta configuración a None para usar cookies basadas en sesión CSRF, que guardan en la memoria las cookies, en vez de usar el almacenamiento persistente.

CSRF_COOKIE_DOMAIN

Valor por omisión: None

El dominio para usar cuando se usa la configuración para cookie CSRF. Esto puede ser útil para fácilmente permitir peticiones de dominios cruzados para excluirlas normalmente de la protección de falsificación de petición de sitio. Esta puede ser una cadena tal como ".example.com", para permitir que una petición POST de un formulario en un subdominio sea validada por una vista que es servida por otro subdominio.

Nota que la presencia de esta configuración no implica que la protección Django CSRF sea segura de ataques en subdominios cruzados de forma predeterminada.

CSRF_COOKIE_HTTPONLY

Valor por omisión: False

Usado solo si utilizas la bandera HttpOnly en la cookie de CSRF. Si se fija en True, Java Script del lado-cliente no podrá acceder a las cookie CSRF.

Esto puede ayudar a prevenir Java Script malicioso que pueda sobrepasar la protección CSRF. Si permites y necesitas enviar valores al CSRF con peticiones Ajax, Java Script necesitara empujar el valor de un token CSRF oculto en los formularios de entrada en la página, en lugar de las cookie.

CSRF_COOKIE_NAME

Valor por omisión: 'csrftoken'

El nombre de la cookie para usar el token CSRF de autentificación. Este puede ser el que quieras.

CSRF_COOKIE_PATH

Valor por omisión: '/'

La ruta establecida en la cookie CSRF. Este debería corresponder a la URL de la ruta de instalación Django o puede ser una ruta padre de esa ruta.

Esto es útil si tienes múltiples instancias de Django ejecutándose bajo el mismo nombre de dominio o `hostname. Puedes usara diferentes rutas para las cookies y cada caso considerará solamente su propia cookies CSRF.

CSRF_COOKIE_SECURE

Valor por omisión: False

Se asegura que la cookie sea marcada como segura. Si está establecido en True, La cookie será marcada como "segura," lo cual quiere decir que los navegadores pueden asegurar que la cookie es sólo enviada bajo una conexión HTTPS.

CSRF_FAILURE_VIEW

Valor por omisión: 'django.views.csrf.csrf_failure'

La ruta a la función vista, para usar cuando una petición entrante sea rechazada por la protección CSRF. La función debe tener esta firma:

```
def csrf_failure(request, reason="")
```

Donde reason es un mensaje corto (previsto para los desarrolladores, no para los usuarios finales) indica la razón por la que la petición fue rechazada.

DATABASES

Valor por omisión: (Un diccionario vacio)

Un diccionario que contiene las configuraciones para todas las bases de datos usadas con Django. Es un diccionario jerarquizado cuyo contenido mapea alias de la base de datos a un diccionario, conteniendo las opciones para una base de datos individual.

La configuración DATABASES debe permitir configurar una base de datos por default y cualquier número de bases de datos adicionales que puedan especificarse.

La configuración más simple posible es para una simple base de datos usando SQLite. Esta se puede configurar de la siguiente forma:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'datos.db',
    }
}
```

Cuando se conecta a otras bases de datos, tal como MySQL, Oracle o PostgreSQL es necesario agregar los parámetros de conexión que requiera.

Por ejemplo para configurar PostgreSQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'datos.db',
        'USER': 'nombreusuario',
        'PASSWORD': 'contraseña',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

Las siguientes opciones internas disponibles pueden ser requeridas para configurar conexiones más complejas usando la variable DATABASES.

ATOMIC_REQUESTS

Valor por omisión: False

Fija este valor a True para envolver cada petición HTTP request en una sola transacción de la base de datos.

AUTOCOMMIT

Valor por omisión: True

Fija este valor a False si quieres desactivar el manejador de transacciones de Django e implementar el tuyo.

ENGINE

Valor por omisión: '' (cadena vacía)

Esta variable de configuración indica qué back-end de base de datos debe usarse, los backends incorporados son:

'django.db.backends.postgresql_psycopg2'
'django.db.backends.mysql'
'django.db.backends.sqlite3'
'django.db.backends.oracle'

Puedes usar una base de datos como backend que no esté listada en Django configurando ENGINE con la ruta completa a tu backend (por ejemplo mipaquete.backends.whatever).

HOST

Valor por omisión: '' (cadena vacía)

Esta variable de configuración indica el dominio debe usarse cuando se establezca una conexión a la base de datos. Una cadena vacía significa localhost. No se usa con SQLite.

Si este valor comienza con una barra (/) y estás usando MySQL, MySQL se conectará al socket vía un socket Unix:

"HOST": '/var/run/mysql'

Si estás usando MySQL este valor *no* comienza con una barra, entonces se asume que el mismo es el nombre del equipo.

NAME

Valor por omisión: '' (cadena vacía)

El nombre de la base de datos a usarse. Para SQLite, es la ruta completa al archivo de la base de datos.

CONN_MAX_AGE

Valor por omisión: 0

El tiempo de vida de la conexión en segundos. Usa 0 para cerrar la conexión a la base de datos al final de cada petición – el comportamiento histórico de Django y None para conexiones persistentes ilimitadas.

OPTIONS

Valor por omisión: {} (Diccionario vacío)

Parámetros extra a usarse cuando se establece la conexión a la base de datos. Los parámetros disponibles varían dependiendo de la base de datos.

PASSWORD

Valor por omisión: '' (cadena vacía)

Esta variable de configuración es la contraseña a usarse cuando se establece una conexión a la base de datos. No se usa con SQLite.

PORT

Valor por omisión: '' (Cadena vacía)

El puerto a usarse cuando se establece una conexión a la base de datos. Una cadena vacía significa el puerto por omisión. No se usa con SQLite.

USER

Valor por omisión: '' (Cadena vacía)

Esta variable de configuración es el nombre de usuario a usarse cuando se establece una conexión a la base de datos. No se usa con SQLite.

TEST

Valor por omisión: {}

Un diccionario de configuraciones para pruebas o test en la base de datos.

DATABASE_ROUTERS

Valor por omisión: [] (Lista vacía)

La lista de routers que pueden usarse para determinar cual base de datos es usada para optimizar las consultas a la base de datos.

DATE_FORMAT

Valor por omisión: 'N j, Y' (por ej. Feb. 4, 2003)

El formato a usar por omisión para los campos de fecha en las páginas lista de cambios en el sitio de administración de Django – y, posiblemente, por otras partes del sistema. Acepta el mismo formato que la etiqueta now.
Ver también DATETIME_FORMAT, TIME_FORMAT, YEAR_MONTH_FORMAT y MONTH_DAY_FORMAT.

DATETIME_FORMAT

Valor por omisión: 'N j, Y, P' (por ej. Feb. 4, 2003, 4 p.m.)

El formato a usar por omisión para los campos de fecha-hora en las páginas lista de cambios en el sitio de administración de Django – y, posiblemente, por otras partes del sistema. Acepta el mismo formato que la etiqueta now ver Apéndice F, Tabla F-2).
Ver también DATE_FORMAT, DATETIME_FORMAT, TIME_FORMAT, YEAR_MONTH_FORMAT y MONTH_DAY_FORMAT.

DATE_INPUT_FORMATS

Valor por omisión:

```
(
   '%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', # '2006-10-25', '10/25/2006', '10/25/06'
   '%b %d %Y', '%b %d, %Y', # 'Oct 25 2006', 'Oct 25, 2006'
   '%d %b %Y', '%d %b, %Y', # '25 Oct 2006', '25 Oct, 2006'
   '%B %d %Y', '%B %d, %Y', # 'Octubre 25 2006', 'Octubre 25, 2006'
   '%d %B %Y', '%d %B, %Y', # '25 Octubre 2006', '25 Octubre, 2006'
)
```

Una tupla de formatos que serán aceptados al introducir datos en un campo date.

DATETIME_FORMAT

Valor por omisión: 'N j, Y, P' (e.g. Feb. 4, 2003, 4 p.m.)

El formato predeterminado para mostrar campos tipo fecha o datetime en cualquier parte del sistema. Nota que si la configuración local USE_L10N es `True esta tendrá mayor precedencia.

DATETIME_INPUT_FORMATS

Valor por omisión:

```
(
    '%Y-%m-%d %H:%M:%S', # '2006-10-25 14:30:59'
    '%Y-%m-%d %H:%M:%S.%f', # '2006-10-25 14:30:59.000200'
    '%Y-%m-%d %H:%M', # '2006-10-25 14:30'
    '%Y-%m-%d', # '2006-10-25'
    '%m/%d/%Y %H:%M:%S', # '10/25/2006 14:30:59'
    '%m/%d/%Y %H:%M:%S.%f', # '10/25/2006 14:30:59.000200'
    '%m/%d/%Y %H:%M', # '10/25/2006 14:30'
    '%m/%d/%Y', # '10/25/2006'
    '%m/%d/%y %H:%M:%S', # '10/25/06 14:30:59'
    '%m/%d/%y %H:%M:%S.%f', # '10/25/06 14:30:59.000200'
    '%m/%d/%y %H:%M', # '10/25/06 14:30'
    '%m/%d/%y', # '10/25/06'
)
```

Una tupla de formatos que serán aceptados al introducir datos en un campo datetime.

DEBUG

Valor por omisión: False

Esta variable de configuración es un Booleano que activa y desactiva el modo de depuración.
Si defines variables de configuración personalizadas, django/views/debug.py tiene una expresión regular HIDDEN_SETTINGS que ocultará de la vista DEBUG todo aquello que contenga SECRET, PASSWORD o PROFANITIES. Esto permite que usuarios en los que no se confía puedan proveer trazas sin ver variables de configuración con contenido importante (u ofensivo).

Sin embargo, nota que siempre existirán secciones de la salida de depuración que son inapropiadas para el consumo del público. Rutas de archivos, opciones de configuración y similares le proveen a potenciales atacantes información extra acerca de tu servidor. Nunca instales un sitio con DEBUG activo.

DEBUG_PROPAGATE_EXCEPTIONS

Valor por omisión: False

Si se establece en True El manejo normal que Django hace de las excepciones de las funciones de vista será suprimido. Esto puede ser útil para algunos tipos de pruebas, asegúrate de solo usarlo en desarrollo.

DECIMAL_SEPARATOR

Valor por omisión: '.' (Punto)

El separador de decimales predeterminado, usado cuando se formatean números decimales. Nota que si la configuración local USE_L10N es `True esta tendrá mayor precedencia.

DEFAULT_CHARSET

Valor por omisión: 'utf-8'

El conjunto de caracteres a usar por omisión para todos los objetos HttpResponse si no se especifica en forma manual un tipo MIME. Se usa en conjunto con DEFAULT_CONTENT_TYPE para construir la cabecera Content-Type.

DEFAULT_CONTENT_TYPE

Valor por omisión: 'text/html'

Tipo de contenido a usar por omisión para todos los objetos HttpResponse, si no se especifica manualmente un tipo MIME. Se usa en conjunto con DEFAULT_CHARSET para construir la cabecera Content-Type. Ver el Apéndice H para conocer más acerca de los objetos HttpResponse.

DEFAULT_EXCEPTION_REPORTER_FILTER

Valor por omisión: django.views.debug.SafeExceptionReporterFilter

Valor predeterminado para el filtro encargado del manejo de reportes de exepciones usado si no se asigna uno.

DEFAULT_FILE_STORAGE

Valor por omisión: django.core.files.storage.FileSystemStorage

Clase de almacenamiento de archivos predeterminado para usar por cualquiera de las operaciones descritas, que no especifiquen un sistema en particular de almacenamiento.

DEFAULT_FROM_EMAIL

Valor por omisión: 'webmaster@localhost'

La dirección de correo a usar por omisión para correspondencia automatizada enviada por el administrador del sitio.

DEFAULT_INDEX_TABLESPACE

Valor por omisión: '' (Cadena vacía)

Predeterminado tablespace para usar como índice en campos que no especifiquen uno, si la base de datos lo soporta.

DEFAULT_TABLESPACE

Valor por omisión: '' (Cadena vacía)

Predeterminado tablespace para usar en modelos que no especifiquen uno, si la base de datos lo soporta.

DISALLOWED_USER_AGENTS

Valor por omisión: () (Tupla vacía)

Una lista de objetos expresiones regulares compiladas que representan cadenas User-Agent que no tiene permitido visitar ninguna página del sitio, a nivel global para el sitio. Usa la misma para bloquear robots y *crawlers* con mal comportamiento. Se usa únicamente si se ha instalado CommonMiddleware (ver *capítulo 17*).

EMAIL_BACKEND

Valor por omisión: 'django.core.mail.backends.smtp.EmailBackend'

El backend usado para enviar.

EMAIL_FILE_PATH

Valor por omisión: No definido

El directorio usado por el backend de emails file para almacenar archivos.

EMAIL_HOST

Valor por omisión: 'localhost'

El host a usarse para enviar e-mail. Ver también EMAIL_PORT.

EMAIL_HOST_PASSWORD

Valor por omisión: '' (cadena vacía)

La contraseña a usarse para el servidor SMTP definido en EMAIL_HOST. Esta variable de configuración se usa en combinación con EMAIL_HOST_USER cuando se está autenticando ante el servidor SMTP. Si alguna de estas variables de configuración está vacía, Django no intentará usar autenticación.

Ver también EMAIL_HOST_USER.

EMAIL_HOST_USER

Valor por omisión: '' (cadena vacía)

El nombre de usuario a usarse para el servidor SMTP definido en EMAIL_HOST. Si está vacío, Django no intentará usar autenticación. Ver también EMAIL_HOST_PASSWORD.

EMAIL_PORT

Valor por omisión: 25

El puerto a usarse pata el servidor SMTP definido en EMAIL_HOST.

EMAIL_SUBJECT_PREFIX

Valor por omisión: '[Django]'

El prefijo del asunto para mensajes de e-mail enviados con django.core.mail.mail_admins o django.core.mail.mail_managers. Probablemente querrás incluir un espacio al final.

FILE_CHARSET

Valor por omisión: 'utf-8'

La codificación del carácteres usada para decodificar cualquier archivo leído del disco. Esto incluye archivos de plantillas y ficheros de datos iniciales SQL.

FILE_UPLOAD_HANDLERS

Valor por omisión:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler")
```

Una tupla de manejadores usados para subir archivos.

FILE_UPLOAD_MAX_MEMORY_SIZE

Valor por omisión: 2621440 (i.e. 2.5 MB).

El tamaño máximo (en bytes) que se permite subir, antes de usar streamed por el sistema de archivos.

FILE_UPLOAD_DIRECTORY_PERMISSIONS

Valor por omisión: None

El modo numérico aplicado a directorios creados en el proceso de subir archivos.

FILE_UPLOAD_TEMP_DIR

Valor por omisión: None

El directorio para almacenar datos (en particular archivos más grandes que FILE_UPLOAD_MAX_MEMORY_SIZE) temporalmente cuando se suben archivos. Si es None, Django usara el directorio temporal usado por el sistema operativo. Por ejemplo, en sistemas estilo-‘nix el valor predeterminado es: /tmp.

FIRST_DAY_OF_WEEK

Valor por omisión: 0 (Domingo)

Numero que representa el primer día de la semana. Especialmente útil cuando se muestra un calendario. Este valor únicamente es usado cuando se usa el formato de internacionalización o cuando no se puede encontrar el actual formato local.

Este valor debe ser un entero entre 0 y 6, donde 0 es Domingo, 1 es lunes y así sucesivamente.

FIXTURE_DIRS

Valor por omisión: () (tupla vacía)

Una lista de ubicaciones para los archivos con datos de fixtures, en el orden en el que se buscará en las mismas. Nota que estas rutas deben usar barras de estilo Unix, aún en Windows.

IGNORABLE_404_ENDS

Valor por omisión: ('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi', 'favicon.ico', '.php')

Lista de expresiones que deben ser ignoradas. Ver IGNORABLE_404_URS.

IGNORABLE_404_URLS

Valor por omisión: ()

Lista de expresiones regulares compiladas que describen las URL que deben ser ignoradas cuando se reportan errores HTTP 404 vía email.

INSTALLED_APPS

Valor por omisión: () (tupla vacía)

Una tupla de cadenas que indican todas las aplicaciones que están activas en esta instalación de
Django. Cada cadena debe ser una ruta completa de Python hacia:

- Una clase para configurar una aplicación, o
- Un paquete que contiene una aplicación.

INTERNAL_IPS

Valor por omisión: () (tupla vacía)

Una tupla de direcciones IP, como cadenas, que:

- Pueden ver comentarios de depuración cuando DEBUG es True.
- Reciben cabeceras X si está instalado XViewMiddleware.

LANGUAGES

Valor por omisión: Una tupla de todos los idiomas disponibles. Esta lista está en continuo crecimiento y cualquier copia que incluyéramos aquí inevitablemente quedaría rápidamente desactualizada. Puedes ver la lista actual de idiomas traducidos examinando django/conf/global_settings.py, o consulta la documentación online disponible en:
https://github.com/django/django/blob/master/django/conf/global_settings.py).

La lista es una tupla de tuplas de dos elementos en el formato (código de idioma, nombre de idioma) – por ejemplo, ('ja', 'Japanese'). Especifica qué idiomas están disponibles para la selección de idioma. Ver el *capítulo 19* para más información acerca de selección de idiomas.
Generalmente, el valor por omisión debería ser suficiente. Solo asigna valor a esta variable de configuración si deseas restringir la selección de idiomas a un subconjunto de los idiomas provistos con Django.

Si asignas un valor personalizado a LANGUAGES, está permitido marcar los idiomas como cadenas de traducción, pero *nunca* debes importar django.utils.translation desde el archivo de configuración, porque ese módulo a su vez depende de la configuración y esto crearía una importación circular.

La solución es usar una función gettext() "boba". A continuación un archivo de configuración ejemplo:

```
gettext = lambda s: s

LANGUAGES = (
   ('de', gettext('German')),
   ('en', gettext('English')),
)
```

Con este esquema, make-messages.py todavía podrá encontrar y marcar esas cadenas para traducción, pero la traducción no ocurrirá en tiempo de ejecución – así que tendrás que recordar envolver los idiomas con la gettext() *real* en todo código que use LANGUAGES en tiempo de ejecución.

MANAGERS

Valor por omisión: () (tupla vacía)

Esta tupla está en el mismo formato que ADMINS que especifica quiénes deben recibir notificaciones de enlaces rotos cuando SEND_BROKEN_LINK_EMAILS tiene el valor True.

MEDIA_ROOT

Default: '' (Empty string) *Valor por omisión*: '' (cadena vacía) La ruta absoluta al directorio del sistema que contiene los archivos subidos por los usuarios

Los valores para MEDIA_ROOT y STATIC_ROOT deben de contener valores distintos.

MEDIA_URL

Valor por omisión: '' (cadena vacía)

Esta URL maneja los medios servidos desde MEDIA_ROOT (por ej. "http://media.lawrence.com").

Nota que esta debe tener una barra final si posee un componente de ruta:

```
Correcto: "http://www.example.com/static/"
Incorrecto: http://www.example.com/static
```

Para usar {{ MEDIA_URL }} en las plantillas, es necesario configurar 'django.core.context_processors.media' en el TEMPLATE_CONTEXT_PROCESSORS.

MIDDLEWARE_CLASSES

Valor por omisión:

```
('django.middleware.common.CommonMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware')
```

Una tupla de clases middleware a usarse. Ver el *capítulo 17*.

MIGRATION_MODULES

Valor por omisión: {} # Un diccionario vacio

Un diccionario que especifica los paquetes donde los módulos de migraciones se pueden encontrar, uno por aplicación. El valor predeterminado de esta configuración es un diccionario vacio, pero el nombre del paquete predeterminado para el modulo de migraciones es migrations.

Ejemplo:

```
{'blog': 'blog.db_migrations'}
```

En este caso, las migraciones relacionadas con la aplicación blog estarán contenidas en el paquete blog.db_migrations

El comando makemigrations automáticamente crea el paquete si este no existe.

MONTH_DAY_FORMAT

Valor por omisión: 'F j'

El formato a usar por omisión para campos de fecha en las páginas de lista de cambios en la aplicación de administración de Django – y, probablemente, en otras partes del sistema – en casos en los que sólo se muestran el mes y el día. Acepta el mismo formato que la etiqueta now.

Por ejemplo, cuando en una página de lista de cambios la aplicación de administración de Django se filtra por una fecha, la cabecera para un día determinado muestra el día y mes. Diferentes locales tienen diferentes formatos. Por ejemplo, el Inglés de EUA tendría "January 1" mientras que Español podría tener "1 Enero".

Ver también DATE_FORMAT, DATETIME_FORMAT, TIME_FORMAT y YEAR_MONTH_FORMAT.

PREPEND_WWW

Valor por omisión: False

Esta variable de configuración indica si se debe agregar el prefijo de subdominio "www." a URLs que no lo poseen. Se usa únicamente si CommonMiddleware está instalado (ver capítulo 17). Ver también APPEND_SLASH.

NUMBER_GROUPING

Valor por omisión: 0

Número de dígitos agrupados juntos en la parte entera de un número.

De uso común para visualizar separadores de mil. Si la configuración es 0, entonces no se aplicara el agrupamiento a los números. Si la configuración es más grande que 0 entonces THOUSAND_SEPARATOR se usara para separar entre esos grupos.

Nota que si el valor de **USE_L10N** está fijado en True, el formato local tendrá precedencia sobre esta configuración.

ROOT_URLCONF

Valor por omisión: No definido

Una cadena que representa la ruta completa de importación Python hacia tu URLconf raíz (por ej. "mydjangoapps.urls"). Ver *capítulo 3*.

STATIC_ROOT

Valor por omisión: None

La ruta absoluta al directorio donde se recolectaran los archivos estáticos para el despliegue, usando el comando collectstatic.

Por ejemplo: "/var/www/example.com/static/"

STATIC_URL

Valor por omisión: None

URL usada para referirse a la ubicación de los archivos estáticos en STATIC_ROOT.
Por ejemplo: "/static/" o http://static.example.com/

SECRET_KEY

Valor por omisión: (Generado automáticamente cuando creas un proyecto)

Una clave secreta para esta instalación particular de Django. Es usada para proveer una semilla para los algoritmos de hashing. Asigna un valor de una cadena con caracteres al azar – mientras más larga mejor. django-admin startproject crea una en forma automática y en la mayoría de los casos no será necesario que la modifiques.

SEND_BROKEN_LINK_EMAILS

Valor por omisión: False

Esta variable de configuración indica si se debe enviar un e-mail a los MANAGERS cada vez que alguien visita una página impulsada por Django que generará un error 404 y que posea una cabecera referir no vacía (en otras palabras un enlace roto). Es solamente usado si está instalado CommonMiddleware (ver *capítulo 17*).

SERIALIZATION_MODULES

Valor por omisión: No definida.

Un diccionario de módulos que contiene las definiciones de serialización (previstas como strings) Con llave para un identificador de cadena para el tipo de serialización. Por ejemplo, para definir un serializador YAML, usa:

SERIALIZATION_MODULES = {'yaml': 'path.to.yaml_serializer'}

SERVER_EMAIL

Valor por omisión: 'root@localhost'

La dirección de e-mail a usarse como remitente para los mensajes de error, tales como los enviados a ADMINS and MANAGERS.

SHORT_DATE_FORMAT

Valor por omisión: m/d/Y (e.g. 12/31/2003)

Un formato disponible que puede usarse para mostrar campos date en las plantillas. Nota que si USE_L10N está fijado en True, el formato local tendrá mayor precedencia y será aplicado.

SHORT_DATETIME_FORMAT

Valor por omisión: m/d/Y P (e.g. 12/31/2003 4 p.m.)

Un formato disponible que puede usarse para mostrar campos datetime en las plantillas. Nota que si USE_L10N está fijado en True, el formato local tendrá mayor precedencia y será aplicado.

SIGNING_BACKEND

Valor por omisión: 'django.core.signing.TimestampSigner'

El backend usado para firma las cookies y otros datos.

SESSION_COOKIE_AGE

Valor por omisión: 1209600 (dos semanas, en segundos)

Esta es la edad de las cookies de sesión, en segundos. Ver *capítulo 14*.

SESSION_COOKIE_DOMAIN

Valor por omisión: None

El dominio a usarse para las cookies de sesión. Asigna como valor una cadena tal como ".lawrence.com" para cookies inter-dominio, o usa None para una cookie de dominio estándar. Ver *capítulo 14*.

SESSION_COOKIE_NAME

Valor por omisión: 'sessionid'

El nombre de la cookie a usarse para las sesiones; puede tener el valor que tu desees. Ver *capítulo 14*.

SESSION_COOKIE_SECURE

Valor por omisión: False

Esta variable de configuración indica si debe usarse una cookie segura para la cookie de sesión. Si tiene un valor True, la cookie será marcada como "segura", lo que significa que los navegadores podrían asegurarse que la cookie sólo se envíe vía una conexión HTTPS. Ver *capítulo 14*.

SESSION_EXPIRE_AT_BROWSER_CLOSE

Valor por omisión: False

Esta variable de configuración indica si las sesiones deben caducar cuando el usuario cierre su navegador. Ver *capítulo 12*.

SESSION_SAVE_EVERY_REQUEST

Valor por omisión: False

Esta variable de configuración indica si la sesión debe ser grabada en cada petición. Ver *capítulo 14*.

SITE_ID

Valor por omisión: No definida.

El identificador, como un entero, del sitio actual en la tabla django_site de la base de datos. Es usada de manera que datos de aplicación puede conectarse en sitio(s) específico(s) y una única base de datos pueda manejar contenido de múltiples sitios. Ver *capítulo 14*.

TEMPLATE_CONTEXT_PROCESSORS

Valor por omisión:

```
("django.contrib.auth.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media",
"django.core.context_processors.static",
"django.core.context_processors.tz",
"django.contrib.messages.context_processors.messages")
```

Una tupla de llamables que son usados para poblar el contexto en RequestContext. Esos llamables reciben como argumento un objeto petición y retornan un diccionario de ítems a ser fusionados con el contexto.

TEMPLATE_DEBUG

Valor por omisión: False

Este Booleano controla el estado encendido/apagado del modo de depuración de plantillas. Si es True la página de error vistosa mostrará un reporte detallado para cada TemplateSyntaxError. Este reporte contiene los fragmentos relevantes de la plantilla, en los cuales se han resaltado las líneas relevantes.

Nota que Django solo muestra páginas de error vistosas si DEBUG es True, así que es posible que desees activar dicha variable para sacar provecho de esta variable.

Ver también DEBUG.

TEMPLATE_DIRS

Valor por omisión: () (tupla vacía)

Un lista de ubicaciones de los archivos de código fuente de plantillas, en el orden en el que serán examinadas. Notar que esas rutas deben usar barras al estilo Unix, aun en Windows. Ver *capítulo 4* y *capítulo 10*.

TEMPLATE_LOADERS

Valor por omisión:

```
('django.template.loaders.filesystem.Loader',
 'django.template.loaders.app_directories.Loader')
```

Una tupla de cargadores de plantillas, especificados como cadenas. Cada clase Loader sabe como importar plantillas desde un particular origen. Opcionalmente, una tupla puede usarse en lugar de una cadena. El primer ítem en la tupla debe ser el modulo Loader los ítems subsecuentes se pasan a Loader durante la inicialización.

TEMPLATE_STRING_IF_INVALID

Valor por omisión: '' (cadena vacía)

La salida, como una cadena, que debe usar el sistema de plantillas para variables inválidas (por ej. con errores de sintaxis en el nombre). Ver Capítulo 10.

TEST_RUNNER

Valor por omisión: 'django.test.simple.run_tests'

El nombre del método a usarse para arrancar la batería de pruebas (por *test suite*). Es usado por el framework de pruebas de Django, el cual se describe en línea en http://www.djangoproject.com/.

TIME_FORMAT

Valor por omisión: 'P' (e.g., 4 p.m.)

El formato a usar por omisión para los campos de hora en las páginas lista de cambios en el sitio de administración de Django – y, posiblemente, por otras partes del sistema. Acepta el mismo formato que la etiqueta now ver Apéndice F, Tabla F-2).

Ver también DATE_FORMAT, DATETIME_FORMAT, TIME_FORMAT, YEAR_MONTH_FORMAT y MONTH_DAY_FORMAT.

TIME_ZONE

Valor por omisión: 'America/Chicago'

Una cadena que representa la zona horaria para esta instalación o None.

Esta es la zona a la cual Django convertirá todas las fechas/horas – no necesariamente la zona horaria del servidor. Por ejemplo, un servidor podría servir múltiples sitios impulsados por Django, cada uno con una configuración de zona horaria separada.

Normalmente, Django fija la variable os.environ['TZ'] a la zona horaria que especificas en la variable de configuración TIME_ZONE. Por lo tanto, todas tus vistas y modelos operarán automáticamente en la zona horaria correcta. Sin embargo, si estás usando el método de configuración manual (descrito arriba en la sección "Usando variables de configuración sin fijar DJANGO_SETTINGS_MODULE") Django *no* tocará la variable de entorno TZ y quedará en tus manos asegurarte de que tus procesos se ejecuten en el entorno correcto.

■ **Nota:** Django no puede usar en forma confiable zonas horarias alternativas en un entorno Windows. Si estás ejecutando Django en Windows debes asignar a esta variable un valor que coincida con la zona horaria del sistema.

USE_ETAGS

Valor por omisión: False

Este Booleano especifica si debe generarse la cabecera ETag. La misma permite ahorrar ancho de banda pero disminuye el rendimiento. Se usa solamente si se ha instalado CommonMiddleware (ver *capítulo 17*).

USE_I18N

Valor por omisión: True

Un Booleano que especifica si debe activarse el sistema de internacionalización de Django (ver *capítulo 19*). Provee una forma sencilla de desactivar la internacionalización, para mejorar el rendimiento. Si se asigna a esta variable el valor False Django realizará algunas optimizaciones de manera que no se cargue la maquinaria de internacionalización.

USE_L10N

Valor por omisión: False

Un Booleano que especifica si debe activarse el sistema de localización de Django (ver *capítulo 19*). Si se fija a True Django mostrara números y fechas usando el formato de la localización actual.
Nota:
El archivo settings.py creado por django-admin startproject incluye por conveniencia USE_L10N = True.

USE_TZ

Valor por omisión: False

Un valor booleano que especifica si se tendrán en cuenta los formatos de fecha y tiempo por defecto o no. Django toma en cuenta los formatos de fechas y tiempos internamente de otra forma Django usara los valores en tiempo local.

■**Nota:** El archivo settings.py creado por django-admin startproject incluye por conveniencia USE_TZ = True.

WSGI_APPLICATION

Valor por omisión: None

La ruta completa al objeto incorporado 'WSGI application' que Django sirve, usando (e.g. runserver) El comando djadmin:*django-admin startproject <startproject>* crea un simple archivo wsgi.py con un llamable llamado application y apunta a este a la configuración de application.

Si no se fija, el valor se usara el valor de retorno de django.core.wsgi.get_wsgi_application(). En este caso el comportamiento de runserver será idéntico al de versiones anteriores de Django.

YEAR_MONTH_FORMAT

Valor por omisión: 'F Y'

El formato a usar por omisión para los campos de fecha en la página de lista de cambios en el sitio de administración de Django – y, posiblemente, por otras partes del sistema, en los casos en los que sólo se muestran el mes y el año. Acepta el mismo formato que la etiqueta *now* ver Apéndice F).

Por ejemplo, cuando se está filtrando una página lista de cambios de la aplicación de administración de Django mediante un detalle de fecha, la cabecera de un mes determinado muestra el mes y el año. Los distintos locales tienen diferentes formatos. Por ejemplo, el inglés de EUA usaría "January 2006" mientras que otro locale podría usar "2006/January".

APÉNDICE E

Etiquetas de plantilla y filtros predefinidos

En el *capítulo 4* se hace una introducción a las etiquetas de plantilla y filtros más utilizados, pero Django incorpora muchos más. En este apéndice se listan toda las que estaban incluidas en el momento en que se escribió el libro, pero se añaden nuevas etiquetas y filtros de forma regular.

La mejor referencia de todas las etiquetas y filtros disponibles se encuentra en la propia página de administración. Allí se incluye una referencia completa de todas las etiquetas y filtros que hay disponibles para una determinada aplicación. Para verla, sólo tienes que pulsar con el ratón en el enlace de documentación que está en la esquina superior derecha de la página.

Las secciones de **etiquetas y filtros** de esta documentación incluirán tanto las etiquetas y filtros predefinidos (de hecho, las referencias de este apéndice vienen directamente de ahí) como aquellas etiquetas y filtros que se hayan incluido o escrito para la aplicación.

Este apéndice será más útil, por tanto, para aquellos que no dispongan de acceso a la interfaz de administración. Como Django es altamente configurable, las indicaciones de la interfaz de administración deben ser consideradas como la documentación más actualizada y, por tanto, la de mayor autoridad.

Etiquetas predefinidas

autoescape

Controla el comportamiento actual del auto-escape. Esta etiqueta toma como argumento tanto a: on y off y determina si el auto-escapeo están dentro del bloque.

Cuando el auto-escapeo esta activado, todas las variables contenidas que contenga HTML serán escapadas antes de mostrar el resultado de la salida (pero después de que cualquier filtro se haya aplicado). Esto es equivalente a manualmente aplicar el filtro escape a cada variable.

La única excepción son las variables que están marcadas como "safe" para autoescape, ya sea por la clave que pobló la variable, o porque se ha aplicado el filtro safe o escape.

Forma de usarlo:

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

block

Define un bloque que puede ser sobrescrito por las plantillas derivadas. Véase la sección acerca de herencia de plantillas en el *capítulo 4* para más información.

comment

Ignora todo lo que aparece entre {% comment %} y {% endcomment %}. Como nota opcional, se puede insertar en la primera etiqueta. Por ejemplo, es útil para comentar fuera del código para documentar, porqué el código fue deshabilitado.

Ejemplo de su uso:

```
<p>Renderizar texto con {{ fecha_publicacion|date:"c" }}</p>
{% comment "Nota opcional" %}
    <p>Comentado fuera del texto {{ creado|date:"c" }}</p>
{% endcomment %}
```

csrf_token

Esta etiqueta es usada para protección CSRF.

cycle

Rota una cadena de texto entre diferentes valores, cada vez que aparece la etiqueta.

Dentro de un bucle, el valor rota entre los distintos valores disponibles en cada iteración del bucle:

```
{% for o in some_list %}
   <tr class="{% cycle row1,row2 %}">
      ...
   </tr>
{% endfor %}
```

Fuera de un bucle, hay que asignar un nombre único la primera vez que se usa la etiqueta, y luego hay que incluirlo ese nombre en las sucesivas llamadas:

```
<tr class="{% cycle row1,row2,row3 as rowcolors %}">...</tr>
<tr class="{% cycle rowcolors %}">...</tr>
<tr class="{% cycle rowcolors %}">...</tr>
```

Se pueden usar cualquier número de valores, separándolos por comas. Asegúrate de no poner espacios entre los valores, sólo comas.

debug

Muestra un montón de información para depuración de errores, incluyendo el contexto actual y los módulos importados.

extends

Sirve para indicar que esta plantilla extiende una plantilla padre.

Esta etiqueta se puede usar de dos maneras:

- {% extends "base.html" %} (Con las comillas) interpreta literalmente "base.html" como el nombre de la plantilla a extender.

- {% extends variable %} usa el valor de variable. Si la variable apunta a una cadena de texto, Django usará dicha cadena como el nombre de la plantilla padre. Si la variable es un objeto de tipo Template, se usará ese mismo objeto como plantilla base.

En él *capítulo 4* podrás encontrar muchos ejemplo de uso de esta etiqueta.

filter

Filtra el contenido de una variable.
Los filtros pueden ser encadenados sucesivamente (La salida de uno es la entrada del siguiente), y pueden tener argumentos, como en la sintaxis para variables
He aquí un ejemplo:

```
{% filter escape|lower %}
  Este texto será escapado y aparecerá en minúsculas
{% endfilter %}
```

firstof

Presenta como salida la primera de las variables que se le pasen que evalúe como no falsa. La salida será nula si todas las variables pasadas valen False.
He aquí un ejemplo:

```
{% firstof var1 var2 var3 %}
```

Equivale a:

```
{% if var1 %}
   {{ var1 }}
{% else %}{% if var2 %}
   {{ var2 }}
{% else %}{% if var3 %}
   {{ var3 }}
{% endif %}{% endif %}{% endif %}
```

for

Itera sobre cada uno de los elementos de un lista o *array*. Por ejemplo, para mostrar una lista de libros, cuyos títulos estén en la lista_libros, podríamos hacer esto:

```
<ul>
   {% for libro in lista_libros %}
      <li>{{ libro.titulo }}</li>
   {% endfor %}
</ul>
```

También se puede iterar la lista en orden inverso usando {% for obj in list reversed %}.

Dentro de un bucle, la propia sentencia for crea una serie de variables. A estas variables se puede acceder únicamente dentro del bucle. Las distintas variables se explican en la Tabla E-1.

Variable	Descripción
forloop.counter	El número de vuelta o iteración actual (usando un índice basado en 1).
forloop.counter0	El número de vuelta o iteración actual (usando un índice basado en 0).
forloop.revcounter	El número de vuelta o iteración contando desde el fin del bucle (usando un índice basado en 1).
forloop.revcounter0	El número de vuelta o iteración contando desde el fin del bucle (usando un índice basado en 0).
forloop.first	True si es la primera iteración.
forloop.last	True si es la última iteración.
forloop.parentloop	Para bucles anidados, es una referencia al bucle externo.

Tabla E1: *Variables accesibles dentro de bucles {% for %}*

for ... empty

La etiqueta for toma una cláusula opcional {% empty %} cuando el texto es mostrado, si el *array* esta vacio o no puede ser encontrado.

```
<ul>
  {% for atleta in lista_atletas %}
    <li>{{ atleta.nombre }}</li>
  {% empty %}
     <li>Lo sentimos, no hay atletas en esta lista.</li>
  {% endfor %}
</ul>
```

El ejemplo anterior es equivalente a (pero más corto, limpio y posiblemente mas rápido) a lo siguiente:

```
<ul>
  {% if lista_atletas %}
     {% for atleta in lista_atletas %}
        <li>{{ atleta.nombre }}</li>
     {% endfor %}
  {% else %}
     <li>Lo sentimos, no hay atletas en esta lista.</li>
  {% endif %}
</ul>
```

If

La etiqueta {% if %} evalúa una variable. Si dicha variable se evalúa como una expresión "verdadera" (Es decir, que el valor exista, no esté vacía y no es el valor booleano False), se muestra el contenido del bloque:

```
{% if lista_atletas %}
```

```
    Número de atletas: {{ lista_atletas|length }}
{% else %}
    No hay atletas.
{% endif %}
```

Si la lista lista_atletas no está vacía, podemos mostrar el número de atletas con la expresión {{ lista_atletas|length }}
Además, como se puede ver en el ejemplo, la etiqueta if puede tener un bloque opcional {% else %} que se mostrará en el caso de que la evaluación de falso.

Operadores booleanos

Las *etiquetas if* pueden usar operadores lógicos como and, or y not para evaluar expresiones más complejas:

```
{% if lista_atletas and lista_entrenadores %}
    Los atletas y los entrenadores están disponibles
{% endif %}

{% if not lista_atletas %}
    No hay atletas.
{% endif %}

{% if lista_atletas or lista_entrenadores %}
    Hay algunos atletas o algunos entrenadores.
{% endif %}

{% if not lista_atletas or lista_entrenadores %}
    No hay atletas o hay algunos entrenadores
{% endif %}

{% if lista_atletas and not lista_entrenadores %}
    Hay algunos atletas y absolutamente ningún entrenador.
{% endif %}
```

La etiqueta if no admite, sin embargo, mezclar los operadores and y or dentro de la misma comprobación, porque la orden de aplicación de los operadores lógicos sería ambigua. Por ejemplo, el siguiente código es inválido:

```
{% if lista_atletas and lista_entrenadores or lista_animadoras %}
```

Para combinar operadores and y or, puedes usar sentencias if anidadas, como en el siguiente ejemplo:

```
{% if lista_atletas %}
    {% if lista_entrenadores or lista_animadoras %}
        ¡Tenemos atletas, y ya sea entrenadores o porristas!
    {% endif %}
{% endif %}
```

Es perfectamente posible usar varias veces un operador lógico, siempre que sea el mismo siempre. Por ejemplo, el siguiente código es válido:

```
{% if lista_atletas or lista_entrenadores or lista_animadoras or lista_profesores %}
```

Las etiquetas if pueden usarse con operadores ==, !=, <, >, <=, >= e in, los cuales funcionan de la siguiente forma:

operador: ==

Igualdad. Por ejemplo:

```
{% if mivariable == "x" %}
    Esta variable aparece si ‘‘mivariable‘‘ es igual a la cadena "x"
{% endif %}
```

operador: !=

Desigualdad. Por ejemplo:

```
{% if mi_variable != "x" %}
   Esta cadena aparece si mi ‘‘mi_variable‘‘ no es igual a la cadena "x",
   o si ‘‘mi_variable‘‘ no se encuentra en el contexto.
{% endif %}
```

operador: <

Menor que. Por ejemplo:

```
{% if mi_variable < 100 %}
    Esta cadena aparece si ‘‘mi_variable‘‘ es menor que 100.
{% endif %}
```

operador: >

Mayor que. Por ejemplo:

```
{% if mi_variable > 0 %}
    Esta cadena aparece si ‘‘mi_variable‘‘‘‘es mayor que 0.
{% endif %}
```

operador: <=

Menor o igual a. Por ejemplo:

```
{% if ‘‘mi_variable‘‘ <= 100 %}
   Esta cadena aparece si ‘‘mi_variable‘‘ es menor o igual a 100.
{% endif %}
```

operador: >=

Mayor o igual a. Por ejemplo:

```
{% if ‘‘mi_variable‘‘ >= 1 %}
   Esta cadena aparece si ‘‘mi_variable‘‘ es mayor o igual que 1.
{% endif %}
```

operador: in

Contenido dentro. Este operador es soportado por muchos contenedores Python para probar si el valor dado está en el contenedor. Los siguientes son algunos ejemplos sobre como x in y son interpretados.

```
{% if "bc" in "abcdef" %}
   Esto aparece si "bc" es una subcadena de "abcdef"
{% endif %}
{% if "hola" in saludos %}
    Si saludos es una lista o un conjunto de elementos, donde uno de los
   elementos de la cadena es "hola", entonces aparecerá.
{% endif %}

{% if usuario in usuarios %}
   Si usuarios es un QuerySet, éste aparecerá si el usuario es un instancia
   que pertenece a el QuerySet.
{% endif %}
```

operador: not in

No contenido dentro. Ésta es la negación del operador in.

El operador de comparación no puede 'encadenar' como en Python en notación Matemática. Por ejemplo en lugar de usar esto:

```
{% if a > b > c %}  (MAL)
```

Debes hacer esto:

```
{% if a > b and b > c %}
```

Filtros

Puedes usar filtros en las expresiones if.

Por ejemplo:

```
{% if messages|length >= 100 %}
   ¡Hoy tienes montones de mensajes!
{% endif %}
```

Expresiones complejas

Todo lo anterior se puede combinar para formar expresiones complejas. Para tales expresiones, puede ser importante saber cómo agrupar los operadores cuando se evalúan las expresiones - es decir, conocer las reglas de prioridad. La precedencia de los operadores, desde lo más bajo a lo más alto posible, es como sigue:

- or
- and
- not
- in
- ==, !=, <, >, <=, >=

(Esto funciona exactamente como en Python). Por ejemplo, la siguiente etiqueta if en Django:

```
{% if a == b or c == d and e %}
```

En Python se escribiría así:

```
(a == b) or ((c == d) and e)
```

Si necesitas usar diferentes prioridades, necesitas jerarquizar las etiquetas if. Algunas veces eso es mejor para obtener mayor claridad, de cualquier manera, es necesario conocer las reglas de precedencia o prioridad.

ifchanged

Comprueba si el valor ha cambiado desde la última iteración al bucle.

El bloque de etiqueta {% ifchanged %} es usado con bucles. Tiene dos posibles usos:

1. Comprueba el contenido dado contra el estado previo y sólo exhibe el contenido si ha cambiado. Por ejemplo, esto muestra una lista de días, únicamente si cambia el mes.

```
<h1>Archivos del {{ year }}</h1>

{% for date in days %}
   {% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}
      <a href="{{ date|date:"M/d"|lower }}/">{{ date|date:"j" }}</a>
{% endfor %}
```

2. Si se le pasan una o más variables, verifica cualquier variable que haya cambiado. El siguiente ejemplo muestra la fecha cada vez que cambia, mientras que muestra la hora si la fecha y la hora cambian:

```
{% for date in days %}
   {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
      {% ifchanged date.hour date.date %}
         {{ date.hour }}
    {% endifchanged %}
{% endfor %}
```

También puede aceptar opcionalmente una cláusula {% else %} que muestra si el valor no ha cambiado.

```
{% for match in matches %}
<div style="background-color:
   {% ifchanged match.ballot_id %}
      {% cycle "red" "blue" %}
   {% else %}
      gray
   {% endifchanged %}
   ">{{ match }}</div>
{% endfor %}
```

ifequal

Muestra el contenido del bloque si los dos argumentos suministrados son iguales.
He aquí un ejemplo:

```
{% ifequal user.id comment.user_id %}
    ...
{% endifequal %}
```

Al igual que con la etiqueta {% if %}, existe una cláusula {% else %} opcional.
Los argumentos pueden ser cadenas de texto, así que el siguiente código es válido:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

Un uso alternativo para la etiqueta ifequal es usarlo con la etiqueta if y el operador ==.

ifnotequal

Es igual que ifequal, excepto que comprueba que los dos parámetros suministrados *no* sean iguales.
Una alternativa para usar la etiqueta ifnotequal es usándola con la etiqueta if y el operador !=.

include

Carga una plantilla y la representa usando el contexto actual. Es una forma de "incluir" una plantilla dentro de otra.
El nombre de la plantilla puede o bien ser el valor de una variable o estar escrita en forma de cadena de texto, rodeada ya sea con comillas simples o comillas dobles, a gusto del lector.
El siguiente ejemplo incluye el contenido de la plantilla "foo/bar.html":

```
{% include "foo/bar.html" %}
```

Este otro ejemplo incluye el contenido de la plantilla cuyo nombre sea el valor de la variable template_name:

```
{% include template_name %}
```

load

Carga una biblioteca de plantillas. En el capítulo 9 puedes encontrar más información acerca de las bibliotecas de plantillas.
Por ejemplo, la siguiente plantilla carga todas las etiquetas y filtros registrados en librería y otra_ librería localizada en el paquete pack:

```
{% load libreria pack.otra_libreria %}
```

También se pueden cargar selectivamente filtros de forma individual o etiquetas de alguna libreria usando el argumento from. En este ejemplo las plantillas de etiquetas/filtros llamada foo y bar serán cargados del paquete algunalibreria:

```
{% load foo bar from algunalibreria %}
```

lorem

Muestra en orden aleatorio el texto en Latin "lorem ipsum". Esto puede ser útil para proveer datos en las plantillas.

Uso:

```
{% lorem [count] [method] [random] %}
```

La etiqueta {% lorem %} puede usarse con cero, uno, dos o tres argumentos. Los argumentos son:

Argumento	Descripción
count	Un numero (o variable) que contiene el numero de párrafos o palabras para generar (el valor predeterminado es 1).
method	Usa w para palabras, p para párrafos en HTML o b para bloques de texto (el valor predeterminado es b).
random	La palabra random, la cual si es dada, no usa el común párrafo ("Lorem ipsum dolor sit amet...") cuando genera texto.

Tabla E2: *Argumentos de la etiqueta lorem.*

Ejemplos:

{% lorem %} la salida con el común párrafo: "lorem ipsum".

{% lorem 3 p %}``la salida con el común párrafo: "lorem ipsum" y dos párrafos al azar en HTML y con etiquetas ``<p>.

{% lorem 2 w random %} la salida serán dos palabras en latín al azar.

now

Muestra la fecha, escrita de acuerdo a un formato indicado.

Esta etiqueta fue inspirada por la función date() de PHP(), y utiliza el mismo formato que esta (http://php.net/date). La versión Django tiene, sin embargo, algunos extras.

He aquí un ejemplo:

```
Es el {% now "jS F Y H:i" %}
```

Se pueden escapar los caracteres de formato con una barra invertida, si se quieren incluir de forma literal. En el siguiente ejemplo, se escapa el significado de la letra "f" con la barra invertida, ya que de otra manera se interpretaría como una indicación de incluir la hora. La "o", por otro lado, no necesita ser escapada, ya que no es un carácter de formato:

```
Es el  {% now "jS o\f F" %}
```

El ejemplo mostraría: "Es el 4th of September".

regroup

Reagrupa una lista de objetos similares usando un atributo común.

Para comprender esta etiqueta, es mejor recurrir a un ejemplo. Digamos que gente es una lista de objetos de tipo Persona, y que dichos objetos tienen los atributos nombre, apellido y género. Queremos mostrar un listado como el siguiente:

- **Hombre:**
  - George Bush
  - Bill Clinton
- **Mujeres:**
  - Margaret Thatcher
  - Condoleezza Rice
- **Desconocido:**
  - Pat Smith

El siguiente fragmento de plantilla mostraría como realizar esta tarea:

```
{% regroup gente by genero as grouped %}
 <ul>
{% for group in grouped %}
   <li>{{ group.grouper }}
      <ul>
        {% for item in group.list %}
           <li>{{ item }}</li>
        {% endfor %}
      </ul>
     </li>
{% endfor %}
</ul>
```

Como puedes ver, {% regroup %} crea una nueva variable, que es una lista de objetos que tienen dos tributos, grouper y list. En grouper se almacena el valor de agrupación, list contiene una lista de los objetos que tenían en común al valor de agrupación. En este caso, grouper podría valer Male, Female y Unknown, y list sería una lista con las personas correspondientes a cada uno de estos sexos.

Hay que destacar que {% regroup %} **no** funciona correctamente cuando la lista no está ordenada por el mismo atributo que se quiere agrupar. Esto significa que si la lista del ejemplo no está ordenada por el sexo, debes asegurarte de que se ordene antes correctamente, por ejemplo con el siguiente código:

```
{% regroup gente|dictsort:"genero" by genero as grouped %}
```

spaceless

Elimina los espacios en blanco entre etiquetas HTML Esto incluye tabuladores y saltos de línea.

El siguiente ejemplo:

```
{% spaceless %}
   <p>
      <a href="foo/">Foo</a>
   </p>
{% endspaceless %}
```

Retornaría el siguiente código HTML:

```
<p><a href="foo/">Foo</a></p>
```

Sólo se eliminan los espacios *entre* las etiquetas, no los espacios entre la etiqueta y el texto. En el siguiente ejemplo, no se quitan los espacios que rodean la palabra Hello:

```
{% spaceless %}
   <strong>
      Hola
   </strong>
{% endspaceless %}
```

ssi

Muestra el contenido de un fichero determinado dentro de la página.

Al igual que la etiqueta "include", {% ssi %} incluye el contenido de otro fichero (que debe ser especificado usando una ruta absoluta) en la página actual:

```
{% ssi /home/html/ljworld.com/includes/right_generic.html %}
```

Si se le pasa el parámetro opcional "parsed", el contenido del fichero incluido se evalúa como si fuera código de plantilla, usando el contexto actual:

```
{% ssi /home/html/ljworld.com/includes/right_generic.html parsed %}
```

Para poder usar la etiqueta {% ssi %}, hay que definir el valor *ALLOWED_INCLUDE_ROOTS* en los ajustes de Django, como medida de seguridad.

La mayor parte de las veces, {% include %} funcionará mejor que {% ssi %}; esta se ha incluido sólo para garantizar compatibilidad hacia atrás.

templatetag

Permite representar los caracteres que están definidos como parte del sistema de plantillas.

Como el sistema de plantillas no tiene el concepto de "escapar" el significado de las combinaciones de símbolos que usa internamente, tenemos que recurrir a la etiqueta {% templatetag %} si nos vemos obligados a representarlos.

Se le pasa un argumento que indica que combinación de símbolos debe producir. Los valores posibles del argumento se muestran en la tabla E-4.

Argumento	Salida
openblock	{%
closeblock	%}
openvariable	{{
closevariable	}}
openbrace	{
closebrace	}
opencomment	{#
closecomment	#}

Tabla F4: *Argumentos válidos de templatetag*

Ejemplo de su uso:

```
{% templatetag openblock %} url 'lista_entradas' {% templatetag closeblock %}
```

url

Devuelve una URL absoluta (Es decir, una URL sin la parte del dominio) que coincide con una determinada vista, incluyendo sus parámetros opcionales. De esta forma se posibilita realizar enlaces sin violar el principio DRY, codificando las direcciones en las plantillas:

```
{% url 'algun-nombre-de-url' v1 v2 %}
```

El primer argumento es el nombre del patrón URL o name. El resto de parámetros son opcionales y deben ir separados con comas, convirtiéndose en parámetros posicionales o por nombre que se incluirán en la URL. Deben estar presentes todos los argumentos que se hayan definido como obligatorios en el URLconf. No es posible mezclar argumentos posicionales y argumentos.

Por ejemplo, supongamos que tenemos una vista, VistaDetallesCliente, y que en el URLconf se la indica que acepta un parámetro, el identificador del cliente. La línea del URL podría ser algo así:

```
url(r'^cliente/(?P<pk>[0-9]+)/$', VistaDetallesCliente.as_view(),
    _name='detalles-cliente'),
```

Si este URLconf fuera incluido en el URLconf del proyecto bajo un directorio, como en este ejemplo:

```
('^clientes/', include('project_name.app_name.urls'))
```

Podríamos crear un enlace a esta vista, en nuestra plantilla, con la siguiente etiqueta:

```
{% url 'detalles-cliente' cliente.id %}
```

La salida de esta etiqueta será /clientes/cliente/123/.

☠**Advertencia:** No olvides poner comillas alrededor del name del patrón, o el valor será interpretado como el contexto de una variable.

Si solo quieres extraer la URL sin mostrarla, puedes usar una llamada un poco diferente:

```
{% url 'algun-nombre_patron' arg arg2 as the_url %}
<a href="{{ the_url }}">Estoy enlazando a {{ the_url }}</a>
```

El alcance de la variable creada por la sintaxis as var es el {% block %} en el cual la etiqueta {% url %} aparece.

Si quieres extraer el namespaced de una URL, especifica la ruta completa a *name* así:
{% url 'miaplicacion:nombre-url' %}

verbatim

Detiene el motor de plantillas que renderiza el contenido de esta etiqueta de bloque.

Un uso muy común es para permitir que Javascript y la capa de plantillas no colisiones con la sintaxis de Django. Por ejemplo:

```
{% verbatim %}
  {{if dying}}Still alive.{{/if}}
{% endverbatim %}
```

También se puede llamar específicamente a una etiqueta de cierre {% endverbatim %} como parte del contenido no renderizado.

```
{% verbatim myblock %}
  Evita la renderizacion a través de {% verbatim %}{% endverbatim %} block.
{% endverbatim myblock %}
```

widthratio

Esta etiqueta es útil para presentar gráficos de barras y similares. Calcula la proporción entre un valor dado y un máximo predefinido, y luego multiplica ese cociente por una constante.

Veamos un ejemplo:

```
<img src="bar.png" alt="Bar"
  height="10" width="{% widthratio this_valor max_valor max_width %}" />
```

Si este valor vale 175 y max_valor es 200, la imagen resultante tendrá un ancho de 88 pixels (porque 175/200 = 0.875 y 0.875 * 100 = 87.5, que se redondea a 88).

En algunos casos es necesario capturar el valor del resultado de widthratio en una variable.
Puede ser útil en instancias, en blocktrans tal como:

```
{% widthratio this_valor max_valor max_width as width %}
{% blocktrans %}The width is: {{ width }}{% endblocktrans %}
```

with

Cachea una variable complicada bajo un nombre más simple. Esto es útil al acceder a un método "costoso" (e.g., uno que 'golpea' la base de datos) varias veces.

Por ejemplo:

```
{% with total=business.employees.count %}
  {{ total }} employee{{ total|pluralize }}
```

```
{% endwith %}
```

La variable poblada (en el ejemplo anterior, total) está únicamente disponible entre las etiquetas {% with %} y {% endwith %}.

Puedes asignar más de una variable al contexto:

```
{% with alpha=1 beta=2 %}
   ...
{% endwith %}
```

Filtros predefinidos

add

Agrega el argumento al valor.

Ejemplo:

```
{{ valor|add:"2" }}
```

Si el valor es 4, la salida será 6.

addslashes

Añade barras invertidas antes de las comillas, ya sean simples o dobles. Es útil para pasar cadenas de texto como javascript, o para escapar cadenas en CVS por ejemplo:

```
{{ valor|addslashes }}
```

Si el valor es "I'm using Django", la salida será: "I\'m using Django"

capfirst

Pasa a mayúsculas la primera letra de la primera palabra. Si el primer carácter no es una letra el filtro no tiene efectos.

Por ejemplo:

```
{{ valor|capfirst }}
```

Si el valor es "django", la salida sea "Django".

center

Centra el texto en un campo de la anchura indicada.

Por ejemplo:

```
"{{ valor|center:"15" }}"
```

Si valor es "Django", la salida será " Django ".

cut

Elimina todos los valores de los argumentos de la cadena dada.

Por ejemplo:

{{ valor|cut:" " }}

Si el valor``es ``"Cadena con espacios", la salida será "Cadenaconespacioss".

date

Formatea una fecha de acuerdo al formato indicado en la cadena de texto (Se usa el mismo formato que con la etiqueta now).

Ejemplo:

{{ valor|date:"F j, Y" }}

Formato	Descripción	Salida
a	'a.m.' o 'p.m.'.	'a.m.' o 'p.m.
A	'AM' o 'PM'.	'AM'
b	El nombre del mes, en forma de abreviatura de tres letras minúsculas.	'jan'
d	Día del mes, dos dígitos que incluyen rellenando con cero por la izquierda si fuera necesario.	'01' a '31'
D	Día de la semana, en forma de abreviatura de tres letras.	'Fri'
f	La hora, en formato de 12 horas y minutos, omitiendo los minutos si estos son cero.	'1', '1:30'
F	El mes, en forma de texto	'January'
g	La hora, en formato de 12 horas, sin rellenar por la izquierda con ceros.	'1' a '12'
G	La hora, en formato de 24 horas, sin rellenar por la izquierda con ceros.	'0' a '23'
h	La hora, en formato de 12 horas.	'01' a '12'
H	La hora, en formato de 24 horas.	'00' a '23'
i	Minutos.	'00' a '59'
j	El día del mes, sin rellenar por la izquierda con ceros.	'1' a '31'
l	El nombre del día de la semana.	'Friday'
L	Booleano que indica si el año es bisiesto.	True o False
m	El día del mes, rellenando por la izquierda con ceros si fuera necesario.	'01' a '12'
M	Nombre del mes, abreviado en forma de abreviatura de tres letras.	'Jan'
n	El mes, sin rellenar con ceros	'1' a '12'
N	La abreviatura del mes siguiendo el estilo de la Associated Press.	'Jan.', 'Feb.' , 'March', 'May'
O	Diferencia con respecto al tiempo medio de Grennwich (*Greenwich Mean Time* - GMT)	'+0200'
P	La hora, en formato de 12 horas, más los minutos, recto si estos son cero y con la indicación a.m./p.m.	'1 a.m.', '1:30 p.m.' , 'midnight',

	Además, se usarán las cadenas de texto especiales 'midnight' y 'noon' para la medianoche y el mediodía respectivamente.	'noon' , '12:30 p.m.'
r	La fecha en formato RFC 822.	'Thu, 21 Dec 2000 16:01:07 +0200'
s	Los segundos, rellenos con ceros por la izquierda de ser necesario.	'00' a '59'
S	El sufijo inglés para el día del mes (dos caracteres).	'st', 'nd', 'rd' o 'th'
t	Número de días del mes.	28 a 31
T	Zona horaria	'EST', 'MDT'
w	Día de la semana, en forma de dígito.	'0' (Domingo) a '6' (Sábado)
W	Semana del año, siguiente la norma ISO-8601, con la semana empezando el lunes.	1, 23
y	Año, con dos dígitos.	'99'
Y	Año, con cuatro dígitos.	'1999'
z	Día del año	0 a 365
Z	Desfase de la zona horaria, en segundos. El desplazamiento siempre es negativo para las zonas al oeste del meridiano de Greenwich, y positivo para las zonas que están al este.	-43200 a 43200

Tabla F-5 *Muestra las cadenas de formato que se pueden utilizar.*

Por ejemplo:

```
{{ valor|date:"D d M Y" }}
```

Si el valor es un objeto datetime (e.g., el resultado de datetime.datetime.now()), la salida será la cadena 'Wed 09 Jan 2008'.

Asumiendo que USE_L10N como True y LANGUAGE_CODE sea, por ejemplo, "es", luego:

```
{{ valor|date:"SHORT_DATE_FORMAT" }}
```

La salida será la cadena "09/01/2008" (en el formato especifico para el local es usando "SHORT_DATE_FORMAT").

default

Si el valor evaluado es False, usa el valor definido como predeterminado o default.

Por ejemplo:

```
{{ valor|default:"nada" }}
```

Si el valor es "" (una cadena vacía), la salida será nada.

default_if_none

Si (y únicamente si) el valor es None, se usa el valor del argumento en su lugar.

Observa que si le pasas una cadena vacía, el valor predeterminado *no* será usado. Usa el fitro default si quieres tratar con cadena vacías.

Por ejemplo:

```
{{ valor|default_if_none:"nada" }}
```

Si valor es None, la salida será una cadena "nada".

dictsort

Acepta una lista de diccionarios y devuelve una lista ordenada según la propiedad indicada en el argumento.

Por ejemplo:

```
{{ valor |dictsort:"nombre" }}
```

Si el valor es:

```
[
  {'nombre': 'zed', 'edad': 19},
  {'nombre': 'amy', 'edad': 22},
  {'nombre': 'joe', 'edad': 31},
]
```

La salida será:

```
[
  {'nombre': 'amy', 'edad': 22},
  {'nombre': 'joe', 'edad': 31},
  {'nombre': 'zed', 'edad': 19},
]
```

Puedes hacer cosas más complicadas como:

```
{% for libro in libros|dictsort:"autor.edad" %}
    {{ libro.titulo }} ({{ libro.autor.nombre }})
{% endfor %}
```

Si libros es:

```
[
  {'titulo': '1984', 'autor': {'nombre': 'George', 'edad': 45}},
  {'titulo': 'Timequake', 'autor': {'nombre': 'Kurt', 'edad': 75}},
  {'titulo': 'Alice', 'autor': {'nombre': 'Lewis', 'edad': 33}},
]
```

La salida será:

- Alice (Lewis)
- 1984 (George)
- Timequake (Kurt)

dictsortreversed

Acepta una lista de diccionarios y devuelve una lista ordenada de forma descendente según la propiedad indicada en el argumento. Trabaja de forma parecida al anterior filtro, pero retorna el valor en orden inverso.

Por ejemplo:

{{ lista|dictsortreversed:"foo" }}

divisibleby

Devuelve True si el valor pasado es divisible por el argumento.

Por ejemplo:

{{ valor|divisibleby:"3" }}

Si el valor es 21, la salida será True.

escape

Escapea una cadena en HTML. Concretamente realiza los siguientes remplazos:

- < es convertido a <
- > es convertido a >
- ' (comillas simples) es convertido a '
- " (comillas dobles) es convertido a "
- & es convertido a &

El escape es únicamente aplicado en la salida de la cadena, así que no importa donde se encadenen la serie de filtros usando escape: este siempre será aplicado como al último filtro. Si quieres que el escape se aplique inmediatamente, utiliza el filtro force_escape.

Aplicar escape a una variable que normalmente esta auto-escapeada, da como resultado que el escapeo se aplique una sola vez. Por lo que es seguro usar esta función incluso en ambientes de auto-escape. Si quieres pasar múltiples escapes usa el filtro force_escape

Por ejemplo, puedes aplicar escape a campos cuando la etiqueta autoescape esta desactivada o en of:

```
{% autoescape off %}
  {{ titulo|escape }}
{% endautoescape %}
```

escapejs

Escapa caracteres para usar en cadenas Javascript. Esta *no* marca las cadenas como seguras para usar en HTML, pero las protege de errores de sintaxis cuando se usan plantillas generadas por JavaScript/JSON.

Por ejemplo:

```
{{ valor|escapejs }}
```

Si valor es "testing\r\njavascript \'string" escaping", la salida será "testing\\u000D\\u000Ajavascript \\u0027string\\u0022 \\u003Cb \\u003Eescaping\\u003C/b\\u003E".

filesizeformat

Representa un valor, interpretándolo como si fuera el tamaño de un fichero y "humanizando" el resultado, de forma que sea fácil de leer. Por ejemplo, las salidas podrían ser '13 KB', '4.1 MB', '102 bytes', etc.

Por ejemplo:

```
{{ valor|filesizeformat }}
```

Si valor es 123456789, la salida será 117.7 MB.
Tamaño de archivos y unidades SI
Estrictamente hablando filesizeformat no se ajusta al Sistema Internacional de Unidades (International System of Units) que recomiende usar KiB, MiB, GiB, cuando el tamaño de los bytes se calcula en torno a 1024(como en este caso). En lugar de eso, Django usa tradicionalmente nombres de unidades (KB, MB, GB, etc.) correspondiendo a los nombres que se utilizan más comúnmente.

first

Devuelve el primer elemento de una lista.

Por ejemplo:

```
{{ valor|first }}
```

Si el valor está en la lista ['a', 'b', 'c'], la salida será 'a'.

floatformat

Si se usa sin argumento, redondea un número en coma flotante a un único dígito decimal (pero sólo si hay una parte decimal que mostrar), por ejemplo:

Valor	Plantilla	Salida
34.23234	{{ valor|floatformat }}	34.2
34.00000	{{ valor|floatformat }}	34
34.26000	{{ valor|floatformat }}	34.3

Si se utiliza un argumento numérico, floatformat redondea a ese número tantos lugares como decimales definidos, por ejemplo:

Valor	Plantilla	Salida
34.23234	{{ valor|floatformat:3 }}	34.232
34.00000	{{ valor|floatformat:3 }}	34.000
34.26000	{{ valor|floatformat:3 }}	34.260

Particularmente útil al pasárselo al 0 (cero) como el argumento que redondea el número flotante, al valor entero más cercano.

Valor	Plantilla	Salida
34.23234	{{ valor|floatformat:"0" }}	34
34.00000	{{ valor|floatformat:"0" }}	34
39.56000	{{ valor|floatformat:"0" }}	40

Si el argumento pasado a floatformat es negativo, redondeará a ese número de decimales, pero sólo si el número tiene parte decimal. Por ejemplo:

Valor	Plantilla	Salida
34.23234	{{ valor|floatformat:"-3" }}	34.232
34.00000	{{ valor|floatformat:"-3" }}	34
34.26000	{{ valor|floatformat:"-3" }}	34.260

Usar floatformat sin argumentos es equivalente a usar floatformat con el argumento de -1.

force_escape

Aplica escapeo HTML a la cadena (consulta el filtro escape para mas detalles). Este filtro es aplicado inmediatamente y devuelve una nueva cadena escapada. Es útil en raros casos, por ejemplo cuando es necesario el uso de múltiples escapeos o cuando es necesario aplicar otro filtro al resultado escapado. Normalmente se usa el filtro escape.

Por ejemplo, si quieres atrapara los elemento HTML <p> creados por el filtro tfilter:*linebreaks*:

```
{% autoescape off %}
  {{ cuerpo|linebreaks|force_escape }}
{% endautoescape %}
```

get_digit

Dado un número, devuelve el dígito que esté en la posición indicada, siendo 1 el dígito más a la derecha. En caso de que la entrada sea inválida, devolverá el valor original (Si la entrada o el argumento no fueran enteros, o si el argumento fuera inferior a 1). Si la entrada es correcta, la salida siempre será un entero.

Por ejemplo:

```
{{ valor|get_digit:"2" }}
```

Si valor es 123456789, la salida será 8.

iriencode

Convierte un IRI (Identificador Internacional de Recursos o Internationalized Resource Identifier) a una cadena que es conveniente para incluir en una URL. Esto es necesario su están tratando de usar cadenas que contienen caracteres que no son ASCII en una URL.

Es seguro usar este filtro en una cadena que ha pasado por un filtro urlencode.

Por ejemplo:

```
{{ valor|iriencode }}
```

Si valor es "?test=1&me=2", la salida será "?test=1&me=2".

join

Concatena todos los elementos de una lista para formar una cadena de texto, usando como separador el texto que se le pasa como argumento. Es equivalente a la llamada en Python str.join(list)

Por ejemplo:

```
{{  valor|join:" // " }}
```

Si valor es la lista ['a', 'b', 'c'], la salida será la cadena: "a // b // c".

last

Devuelve el último ítem de una lista.

Por ejemplo:

```
{{ valor|last }}
```

Si valor es la lista ['a', 'b', 'c', 'd'], la salida será la cadena "d".

length

Devuelve la longitud del valor. Funciona tanto en listas como en cadenas.

Por ejemplo:

```
{{ valor|length }}
```

Si el valor es ['a', 'b', 'c', 'd'] o "abcd", la salida será 4.
El filtro devuelve 0 cuando las variables no están definidas.

length_is

Devuelve el valor True si la longitud de la entrada coincide con el argumento suministrado, o de lo contrario False.

Por ejemplo:

```
{{ valor|length_is:"4" }}
```

Si el valor es ['a', 'b', 'c', 'd'] o "abcd", la salida será True.

linebreaks

Remplaza saltos de línea en texto plano con los apropiados formatos en HTML; una simple nueva línea se convierte en un salto de línea en HTML (
) y una nueva línea seguida de una línea en blanco se convierte en un párrafo (</p>).

Por ejemplo:

```
{{ valor|linebreaks }}
```

Si el valor es Joel\nes un slug, la salida será Joel
es un slug .

linebreaksbr

Convierte todos los saltos de línea en etiquetas
.

Por ejemplo:

```
{{ valor|linebreaksbr }}
```

Si el valor es Joel\nes un slug, la salida será Joel
es un slug.

linenumbers

Muestra el texto de la entrada con números de línea.

Por ejemplo:

```
{{ valor|linenumbers }}
```

Si el valor es:

```
uno
dos
tres
```

La salida será:

1. uno
2. dos
3. tres

ljust

Justifica el texto de la entrada a la izquierda utilizando la anchura indicada.

Argumento: tamaño de campo

Por ejemplo:

```
{{ valor|ljust:"10" }}
```

Si el valor is Django, la salida será "Django ".

lower

Convierte el texto de la entrada dada, a letras en minúsculas

Por ejemplo:

```
{{ valor|lower }}
```

Si el valor es Sigo ENOJADO con Yoko, la salida será sigo enojado con yoko.

make_list

Devuelve la entrada en forma de lista. Si la entrada es un número entero, se devuelve una lista de dígitos. Si es una cadena de texto, se devuelve una lista de caracteres.

Por ejemplo:

```
{{ valor|make_list }}
```

Si el valor es la cadena "Joel", la salida será la lista: ['J', 'o', 'e', 'l']. Si el valor es 123, la salida será la lista: ['1', '2', '3']

phone2numeric

Convierte un número de teléfono (que incluso puede contener letras) a su forma numérica equivalente.

La entrada no tiene porque ser un número de teléfono válido. El filtro convertirá alegremente cualquier texto que se le pase.

Por ejemplo:

```
{{ valor|phone2numeric }}
```

Si el valor es 800-COLLECT, la salida será: 800-2655328.

pluralize

Retorno el sufijo para formar el plural de cualquier palabra, si el valor es mayor que uno. Por defecto el sufijo es 's'.

Ejemplo:

```
Tú tienes {{ num_mensajes }} mensaje {{ num_mensajes|pluralize }}.
```

Si num_mensajes es 1, la salida será Tu tienes 1 mensaje. Si num_mensajes es 2, la salida será Tu tienes 2 mensajes.

Para aquellas palabras que requieran otro sufijo para formar el plural, podemos usar una sintaxis alternativa en la que indicamos el sufijo que queramos con un argumento.

Ejemplo:

```
Hay registrados {{ num_autores }} autor{{ num_autores|pluralize:"es" }}.
```

Para aquellas palabras que forman el plural de forma más compleja que con un simple sufijo, hay disponible una opción, que permite indicar las formas en singular y en plural, separándolas con una coma.

Ejemplo:

```
Tú tienes{{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

Usa la etiqueta blocktrans para pluralizar cadenas traducidas.

pprint

Un contenedor que permite llamar a la función de Python pprint.pprint. Se usa sobre todo para tareas de depurado de errores.

Ejemplo:

```
{{ objeto|pprint }}
```

random

Devuelve un elemento elegido al azar de la lista.

Por ejemplo:

```
{{ valor|random }}
```

Si el valor es la lista ['a', 'b', 'c', 'd'], la salida podría ser: "b".

rjust

Justifica el texto de la entrada a la derecha utilizando la anchura indicada.
Argumento: El tamaño del campo.
Por ejemplo:

```
"{{ valor|rjust:"10" }}"
```

Si el valor es Django, la salida será "   Django".

safe

Marca una cadena como no requerida para escapeo antes de la salida en HTML. Cuando el autoescape está en of, este filtro no tiene efecto.

■**Nota:** Si estas encadenando filtros, un filtro aplicado después de safe puede hacer el contenido inseguro otra vez. Por ejemplo, el siguiente código imprime las variables como si no estuvieran escapadas:

```
{{ varible|safe|escape }}
```

Aplica el filtro safe a cada elemento de una secuencia. Útil en conjunto con otros filtros que operan en secuencia, tal como el filtro join.

Por ejemplo:

```
{{ alguna_lista|safeseq|join:", " }}
```

No se puede usar el filtro safe directamente en este caso, es necesario convertir la variable en una cadena, en vez de trabajar con lo los elementos individuales de la secuencia.

slice

Devuelve una sección de la lista.

Usa la misma sintaxis que se usa en Python para seccionar una lista. Véase: http://www.diveintopython3.net/native-datatypes.html#slicinglists para una rápida introducción.

Ejemplo:

```
{{ una_lista|slice:":2" }}
```

Si una_lista es ['a', 'b', 'c'], la salida será: ['a', 'b'].

slugify

Convierte a ASCCI. Convierte el texto a minúsculas, elimina los caracteres que no formen palabras (caracteres alfanuméricos y carácter subrayado), y convierte los espacios en guiones.

También elimina los espacios que hubiera al principio y al final del texto.

Por ejemplo:

```
{{ valor|slugify }}
```

Si el valor es "Joel es un slug", la salida será "joel-es-un-slug".

stringformat

Formatea el valor de entrada de acuerdo a lo especificado en el formato que se le pasa como parámetro. La sintaxis a utilizar es idéntica a la de Python, con la excepción de que el carácter "%" se omite.

Puedes consultar las opciones de formateo de cadenas de Python: en http://docs.python.org/library/stdtypes.html#string-formatting-operations para más detalles.

Por ejemplo:

```
{{ valor|stringformat:"E" }}
```

Si el valor es 10, la salida será 1.000000E+01.

striptags

Hace todo los posible por eliminar todas las etiquetas [X]HTML.

Por ejemplo:

```
{{ valor|striptags }}
```

Si el valor es "Joel <button>es</button> un slug", la salida será: "Joel es un slug".

No se garantiza que sea seguro.

Nota que striptags no ofrece ninguna garantía acerca de la salida segura en HTML, en particular con entradas no validas de HTML. Por lo que **NUNCA** apliques el filtro safe a la salida de striptags. Si estas buscando algo más robusto usa la librería Python bleach, en especial el método clean .

time

Formateas una fecha de acuerdo al formato dado.

El formato puede ser predefinido con TIME_FORMAT, o con un formato personalizado, al igual que el filtro date. Nota que el formato predefinido es dependiente del valor local.

Por ejemplo:

```
{{ valor|time:"H:i" }}
```

Si el valor es equivalente a *datetime.datetime.now(),* la salida será la cadena "01:23".

Otro ejemplo:

Asumiendo que USE_L10N sea True y LANGUAGE_CODE sea, por ejemplo "de", entonces para:

```
{{ valor|time:"TIME_FORMAT" }}
```

La salida será la cadena "01:23:00" (El formato especifico "TIME_FORMAT" para el valor local de en Django es "H:i:s")

El filtro time únicamente acepta parámetros en el formato de cadenas o strings que se relacionen con la hora, no con la fecha (por obvias razones). Si necesitas un formato para valores date usa el filtro date en su lugar.

Hay una excepción a la regla anterior: Cuando se pasa un valor datetime con información adjunta timezone, el filtro time acepta el formato timezone específicamente los formatos 'e', 'O' , 'T' y 'Z'.

Cuando se usa sin un formato de cadenas:

```
{{ valor|time }}
```

El formato definido con TIME_FORMAT será usado si se aplica la localización.

timesince

Formatea una fecha como un intervalo de tiempo (por ejemplo, “4 días, 6 horas”).

Acepta un argumento opcional, que es una variable con la fecha a usar como punto de referencia para calcular el intervalo (Si no se especifica, la referencia es el momento *actual*). Por ejemplo, si blog_date es una fecha con valor igual a la medianoche del 1 de junio de 2006, y comment_date es una fecha con valor las 08:00 horas del día 1 de junio de 2006, entonces lo siguiente devolverá “8 horas”.

```
{{ blog_date|timesince:comment_date }}
```

Los minutos son la unidad más pequeña usada y “0 minutos” será devuelto por cualquier fecha que este en el futuro con relación al punto de comparación.

timeuntil

Es similar a timesince, excepto en que mide el tiempo desde la fecha de referencia hasta la fecha dada. Por ejemplo, si hoy es 1 de junio de 2006 y conference_date es una fecha cuyo valor es igual al 29 de junio de 2006, entonces {{ conference_date|timeuntil:from_date }} devolverá "4 semanas".

Acepta un argumento opcional, que es una variable con la fecha a usar como punto de referencia para calcular el intervalo, si se quiere usar otra distinta del momento *actual*. Si from_date apunta al 22 de junio de 2006, entonces {{ conference_date|timeuntil:from_date }} devolvera "1 semana".

```
{{ conference_date|timeuntil:from_date }}
```

Los minutos son la unidad más pequeña usada y "0 minutos" será devuelto por cualquier fecha que este en el futuro con relación al punto de comparación.

title

Convierte una cadena de texto en forma de título, siguiendo las convenciones del idioma inglés (todas las palabras con la inicial en mayúscula).

Por ejemplo:

```
{{ valor|titulo }}
```

Si el valor es "mi PRIMER post", la salida será "Mi primer Post".

truncatechars

Recorta la salida de una cadena de forma que tenga como máximo el número de caracteres que se indican en el argumento. Las cadenas truncadas terminarán con una secuencia de puntos de suspensión ("...").

Argumento: El numero de caracteres a recortar.
Por ejemplo:

```
{{ valor|truncatechars:9 }}
```

Si el valor es "Joel es un slug", la salida será "Joel e...".

truncatechars_html

Parecida al filtro truncatechars, excepto que es capaz de reconocer las etiquetas HTML y, por tanto, no deja etiquetas "huérfanas". Cualquier etiqueta que se hubiera abierto antes del punto de recorte es cerrada por el propio filtro.

Por ejemplo:

```
{{ valor|truncatechars_html:9 }}
```

Si el valor es " Joel es un slug ", la salida será: " Joel i... ".
Las nuevas líneas en el contenido del HTML serán conservadas.

truncatewords

Recorta la salida de forma que tenga como máximo el número de palabras que se indican en el argumento.

Argumento: El numero de palabras a recortar o truncar
Por ejemplo:

```
{{ valor|truncatewords:2 }}
```

Si el valor es "Joel es un slug", la salida será: "Joel es ...".
Las nuevas líneas en la cadena serán removidas.

truncatewords_html

Es similar a truncatewords, excepto que es capaz de reconocer las etiquetas HTML y, por tanto, no deja etiquetas "huérfanas". Cualquier etiqueta que se hubiera abierto antes del punto de recorte es cerrada por el propio filtro.

Es menos eficiente que truncatewords, así que debe ser usado solamente si sabemos que en la entrada va texto HTML.Por ejemplo:

```
{{ valor|truncatewords_html:2 }}
```

Si el valor es " Joel es un slug ", la salida será: " Joel es ... ".
Las nuevas líneas en el contenido del HTML serán conservadas.

upper

Convierte una cadena o string a mayúsculas.

Por ejemplo:

```
{{ valor|upper }}
```

Si el valor es "Joel es un slug", la salida será: "JOEL ES UN SLUG".

urlencode

Escapa la entrada de forma que pueda ser utilizado dentro de una URL.

Por ejemplo:

```
{{ valor|urlencode }}
```

Si el valor es "http://www.example.org/foo?a=b&c=d", la salida será: "http%3A//www.example.org/foo%3Fa%3Db%26c%3Dd".

Opcionalmente podemos pasarle un argumento que contiene los carácteres que no deben ser escapados. Si no se le provee, el carácter '/' es asumido como seguro. Una cadena vacía puede proveerse cuando *todos* los caracteres deben ser escapados. Por ejemplo:

```
{{ valor|urlencode:"" }}
```

Si el valor es “http://www.example.org/”, la salida será: “http%3A%2F%2Fwww.example.org%2F”.

urlize

Convierte URLs y direcciones de email en texto a enlaces HTML.

Esta plantilla de etiqueta funciona en enlaces que contienen prefijos como http://, https://, o www.. Por ejemplo, http://goo.gl/aia1t será convertido, pero goo.gl/aia1t no.

También soporta links, únicamente en el nivel superior de dominios (.com, .edu, .gov, .int, .mil, .net, y .org). Por ejemplo funciona con djangoproject.com.

Por ejemplo:

```
{{ valor|urlize }}
```

Si valor es “Check out www.djangoproject.com”, la salida será “Check out www.djangoproject.com”.

urlizetrunc

Convierte las direcciones URL de un texto en enlaces al igual que urlize, recortando la representación de la URL para que el número de caracteres sea como máximo el del argumento suministrado.

Argumento: Numero de caracteres que el enlace de texto debe contener incluyendo los puntos suspensivos que agrega el filtro.

Por ejemplo:

```
{{ valor|urlizetrunc:15 }}
```

Si el valor es “Check out www.djangoproject.com”, la salida será: “Check out www.djangopr...”.
Al igual que urlize, solo se puede aplicar al texto plano.

wordcount

Devuelve el número de palabras en la entrada.

Por ejemplo:

```
{{ valor|wordcount }}
```

Si el valor es "Joel es un slug", la salida será 4.

wordwrap

Ajusta la longitud del texto para las líneas se adecúen a la longitud especificada como argumento.

Argumento: El numero de caracteres a los cuales ajustar el texto.

Por ejemplo:

```
{{ valor|wordwrap:5 }}
```

Si el valor es Joel es un slug, la salida será:

```
Joel
is a
slug
```

yesno

Dada una serie de textos que se asocian a los valores de True, False y (opcionalmente) None, devuelve uno de esos textos según el valor de la entrada. Véase la siguiente tabla:

Por ejemplo:

```
{{ valor|yesno:"yeah,no,maybe" }}
```

Valor	Argumento	Salida
True	"yeah,no,maybe"	yeah
False	"yeah,no,maybe"	no
None	"yeah,no,maybe"	maybe
None	"yeah,no"	"no" (considera None como False si no se asigna ningún texto a None.

Tabla E6 *Ejemplos del filtro yesno*

Filtros y etiquetas de internacionalización

Django proporciona una serie de etiquetas y filtros que controlan cada aspecto de la Internacionalización en las plantillas. Permiten mantener de forma granular las traducciones, el formato, y las conversiones de las zonas horarias.

i18n

Esta libreria permite especificar el texto traducible en las plantillas. Para usarla asegúrate que USE_I18N este establecido en True, luego cárgala con {% load i18n %}.

l10n

Esta libreria proporciona control sobre los valores de localización en las plantillas. Únicamente necesitas cargar esta libreria usando {% load l10n %}, pero necesitas que USE_L10N a True de modo que la localización este activa de forma predeterminada.

tz

Esta libreria proporciona control sobre las conversiones de zonas horario en las plantillas, tal como l10n, únicamente necesitas cargar la libreria usando {% load tz %}

pero necesitas que USE_TZ sea True de modo que la conversión este activa de forma predeterminada.

Otras etiquetas y filtros útiles

Django viene además con unas par de bibliotecas, que permiten usar otras etiquetas, que es necesario habilitar explícitamente en INSTALLED_APPS para usarlas en las plantillas con la etiqueta {% load %}.

django.contrib.humanize

Un conjunto de filtros de plantillas, útiles para darle un toque humano a los datos.

django.contrib.webdesign

Una colección de etiquetas de plantilla, que pueden ser útiles en el diseño de sitios web, tal como la generación de texto del tipo: Lorem Ipsum.

static

Para enlazar los archivos estáticos que se guardan en STATIC_ROOT Django usa la etiqueta static. Puedes usar esta etiqueta de plantilla independientemente de que uses RequestContext o no.

```
{% load static %}
<img src="{% static "imagenes/hola.jpg" %}" alt="Hola!" />
```

Puedes usarla para enlazar y consumir diferentes variables de contexto, además de los estándares, por ejemplo asumiendo que la variable hoja_de_estilo.css sea pasada a la plantilla:

```
{% load static %}
<link rel="stylesheet" href="{% static hoja_de_estilo.css %}"
type="text/css" media="screen" />
```

Si lo que quieres es recuperar la URL estática para mostrarla, puedes usar una llamada un poco diferente:

```
{% load static %}
{% static "imagenes/hola.jpg" as mifoto %}
<img src="{{ mifoto }}"></img>
```

get_static_prefix

Es preferible que uses la etiqueta de plantilla static, pero si necesitas tener un control más exacto sobre cómo y dónde STATIC_URL es inyectado en la plantilla, puedes usar la etiqueta de plantilla: get_static_prefix:

```
{% load static %}
<img src="{% get_static_prefix %}imagenes/hi.jpg" alt="Hi!" />
```

Existe una segunda forma de poder utilizarla y así evitar el procesamiento extra si necesitas usar varias veces, múltiples valores:

```
{% load static %}

{% get_static_prefix as STATIC_PREFIX %}
<img src="{{ STATIC_PREFIX }}imagenes/hi.jpg" alt="Hi!" />
<img src="{{ STATIC_PREFIX }}imagenes/hi2.jpg" alt="Hola!" />
```

get_media_prefix

Parecida a la etiqueta get_static_prefix, get_media_prefix puebla la variable de la plantilla con el prefijo de media MEDIA_URL.

Por ejemplo:

```
<script type="text/javascript" charset="utf-8">
  var media_path = '{% get_media_prefix %}';
</script>
```

APÉNDICE F

El utilitario django-admin

El utilitario de línea de comandos de Django, es **django-admin.py** diseñado para realizar tareas administrativas. Este apéndice explica sus múltiples poderes.

Usualmente accedes a django-admin.py a través del wrapper del proyecto manage.py, **manage.py** es creado automáticamente en cada proyecto Django y es un contenedor liviano en torno a django-admin.py. Toma cuidado de algunas cosas por ti antes de delegar el trabajo a django-admin.py:

- Pone el paquete de tu proyecto en sys.path.
- Establece la variable de entorno DJANGO_SETTINGS_MODULE para que apunte al archivo settings.py de tu proyecto.
- Llama a django.setup() para inicializar varias funciones internas de Django.

El script django-admin.py debe estar en la ruta de tu sistema si instalaste Django mediante su utilitario setup.py. Si no está en tu ruta, puedes encontrarlo en site-packages/django/bin dentro de tu instalación de Python. Considera establecer un enlace simbólico a él desde algún lugar en tu ruta, como en /usr/local/bin.

Los usuarios de Windows, que no disponen de la funcionalidad de los enlaces simbólicos, pueden copiar django-admin.py a una ubicación que esté en su ruta existente o editar la configuración del PATH (bajo Configuración > Panel de Control > Sistema > Avanzado > Entorno) para apuntar a la ubicación de su instalación.

Generalmente, cuando se trabaja en un proyecto Django simple, es más fácil usar manage.py. Usa django-admin.py con DJANGO_SETTINGS_MODULE o la opción de línea de comando --settings, si necesitas cambiar entre múltiples archivos de configuración de Django.

Uso

```
django-admin.py  <comando> [opciones]
```

O puedes usar:

```
manage.py <comando> [opciones]
```

Donde comando debe ser uno de los comandos listados en este documento y las opciones, son opcionales y pueden ser cero o más de las listadas en este documento.

Los ejemplos de línea de comandos a lo largo de este apéndice usan django-admin.py para ser consistentes, pero cada ejemplo puede usar de la misma forma manage.py. (O simplemente django-admin o manage sin la extencion .py, dependiendo de tu tipo de instalación.)

Obtener ayuda

django-admin. py help

Ejecuta django-admin help para mostrar la información de uso y una lista de todos los comandos proporcionados por cada aplicación.

Ejecuta django-admin help --commands para mostrar una lista de todos los comandos disponibles.

Ejecuta django-admin help <command> para mostrar la descripción de un comando dado y una lista de las opciones disponibles.

Nombres de las aplicaciones

Muchos de los subcomandos toman una lista de "nombres de aplicaciones". Un nombre de una aplicación es el nombre base del el paquete que contiene el modelo. Por ejemplo, si tu INSTALLED_APPS contiene la cadena 'misitio.blog', el nombre de la aplicación es blog.

Determinar la versión

django-admin.py versión

Ejecuta *django-admin.py --version* para mostrar la versión actual de Django.

Ejemplos de la salida:

- 1.7
- 1.8
- 1.9

Mostrar la salida de depuración

Usa --verbosity para especificar la cantidad de notificaciones e información de depuración que django-admin.py debe imprimir en consola. Para más información consulta la documentación de la opción --verbosity.

Subcomandos Disponibles

Las siguientes secciones cubren las acciones disponibles.

check <appname appname ...>

django-admin .py check

Usa el framework check para inspeccionar el proyecto completo para detectar problemas comunes.

El framework check Confirma que no haya ningún problema con los modelos instalados o los registros en la interfaz administrativa. También provee de advertencias para detectar problemas comunes de compatibilidad, introducidos al actualizar Django a una nueva versión. También se pueden realizar chequeos personalizados usando otras bibliotecas y otras aplicaciones.

De forma predeterminada, todas las aplicaciones serán checadas. Puedes checar un conjunto de aplicaciones proporcionando una lista, de cada una de las aplicaciones como argumentos:

```
python manage.py check auth admin myapp
```

Si no especificas alguna aplicación, todas las aplicaciones serán checadas.

--tag <tagname>

El framework check realiza diferentes tipos de chequeos. Estos tipos de chequeos están clasificados en diferentes categorías, agrupadas en etiquetas. Puedes usar estas etiquetas para restringir el chequeo realizado a una categoría en específico. Por ejemplo para únicamente realizar un chequeo de seguridad y compatibilidad, puedes ejecutar:

```
python manage.py check  --tag security  --tag compatibility
```

--list-tags

Para obtener una lista de todas las etiquetas de categorías disponibles, usa:

--deploy

La opción --deploy activa una serie de chequeos adicionales, que son relevantes únicamente en configuración de producción.

Compilemessages

django-admin.py compilemessages

Compila archivos .po creado s por makemessages a archivos .mo para ser usados por el soporte gettext.

Usa la opción --locale (o su version corta -l) para especificar la localidad(es) a procesar. Si no la provees, todas las localidades serán procesadas.

Usa la opción --exclude (o su version corta -x) para especificar la localidad(es) a excluir del procesamiento. Si no la provees ninguna localidad será excluida.

Ejemplos de su uso:

```
django-admin compilemessages --locale=pt_BR
django-admin compilemessages --locale=pt_BR --locale=fr
django-admin compilemessages -l pt_BR
django-admin compilemessages -l pt_BR -l fr
django-admin compilemessages --exclude=pt_BR
django-admin compilemessages --exclude=pt_BR --exclude=fr
django-admin compilemessages -x pt_BR
django-admin compilemessages -x pt_BR -x fr
```

createcachetable

django-admin.py createcachetable

Crea una tabla de cache llamada tablename para usar con el back-end de cache de la base de datos. Ver el capítulo 15 para más información sobre la cache.

La opción --database puede ser usada para especificar la base de datos en la cual se instalara la tabla de cache. Sin embargo no es necesario proveer el nombre para la tabla de la cache a la opción --database. Django toma esta información del archivo de configuración. Si tienes configuradas múltiples caches o múltiples bases de datos, todas las tablas de cache serán creadas.

dbshell

django-admin.py dbshell

Corre el cliente de línea de comandos del motor de base de datos especificado en tu configuración de DATABASE_ENGINE, con los parámetros de conexión especificados en la configuración de DATABASE_USER, DATABASE_PASSWORD, etc.

- Para PostgreSQL, esto ejecuta el cliente de línea de comandos psql .
- For MySQL, esto ejecuta el cliente de línea de comandos mysql.
- For SQLite, esto ejecuta el cliente de línea de comandos sqlite3.

Este comando asume que los programas están en tu PATH de manera que una simple llamada con el nombre del programa (psql, mysql, o sqlite3) encontrará el programa en el lugar correcto. No hay forma de especificar en forma manual la localización del programa.

La opción --database puede usarse para especificar la base de datos sobre la cual abrir el Shell de comandos.

diffsettings

django-admin.py diffsettings

Muestra las diferencias entre la configuración actual y la configuración por omisión de Django.

Las configuraciones que no aparecen en la configuración por omisión están seguidos por "###".

Por ejemplo, la configuración por omisión no define ROOT_URLCONF, por lo que si aparece ROOT_URLCONF en la salida de diffsettings lo hace seguido de "###".

La opción --all puede ser usada para mostrar todas las configuraciones, incluso si tienen valores predefinidos por Django. Tales configuraciones aparecen seguidas del prefijo de "###".

Observa que la configuración por omisión de Django habita en django.conf.global_settings, si alguna vez sientes curiosidad por ver la lista completa de valores por omisión.

dumpdata [appname appname ...]

django-admin.py dumpdata

Dirige a la salida estándar todos los datos de la base de datos asociados con la(s) aplicación(es) nombrada(s).

Si no se le provee el nombre de una aplicación, todas las aplicaciones instaladas serán volcadas.

La salida de dumpdata puede ser usada como entrada para loaddata. Observa que dumpdata usa el manager predeterminado en el modelo para seleccionar el volcado. Si estas usando un manager personalizado como el manejador

predeterminado y si filtras algunos registros disponibles, no todos los objetos serán volcados.

La opción --all puede ser usada para especificar el manejador base que debería usar dumpdata como manager, para volcar registros los cuales han sido filtrados o modificados por un manager personalizado.

--format <fmt>
--indent <num>

Por omisión, la base de datos será volcada en formato JSON. Si quieres que la salida esté en otro formato, usa la opción --format (ej.: format=xml). Puedes especificar cualquier back-end de serialización de Django (incluyendo cualquier back-end de serialización especificado por el usuario mencionado en la configuración de SERIALIZATION_MODULES setting).

De forma predeterminada la salida de dumpdata se da en una simple línea. Que no es sencilla de leer, puedes usar la opción --indent para mostrar la salida indentada de acuerdo a el numero de espacios.

La opción --exclude puede ayudar a prevenir que especificas aplicaciones o modelos (especificados en el formato app_label.ModelName) sean volcadas. Si especificas un nombre de un modelo a dumpdata, la salida del volcado será restringida a ese modelo, en lugar de la aplicación. Puedes mezclar nombres de aplicaciones y nombres de modelos.

flush

django-admin.py flush

Remueve todos los datos de la base de datos, esto significa que todos los datos serán eliminados de la base de datos, todo manejador de post-sincronización será reejecutado, y los datos iníciales serán reinstalados.

La opción --noinput puede proveerse para suprimir todos los mensajes de confirmación de los comandos en la terminal del usuario.

La opción --database puede ser usada para especificar la base de datos a vaciar (flush)

--no-initial-data

Usa --no-initial-data para evitar cargar en la instalación datos iníciales.

inspectdb

django-admin.py inspectdb

Realiza la introspección sobre las tablas de la base de datos apuntada por la configuración NAME y envía un modulo de modelo de Django (un archivo models.py) a la salida estándar.
Usa esto si tienes una base de datos personalizada con la cual quieres usar Django. El script inspeccionará la base de datos y creará un modelo para cada tabla que contenga.

Como podrás esperar, los modelos creados tendrán un atributo por cada campo de la tabla. Observa que inspectdb tiene algunos casos especiales en los nombres de campo resultantes:

- Si inspectdb no puede mapear un tipo de columna a un tipo de campo del modelo, usará TextField e insertará el comentario Python 'This field type is a guess.' junto al campo en el modelo generado.

- Si el nombre de columna de la base de datos es una palabra reservada de Python (como 'pass', 'class', o 'for'), inspectdb agregará '_field' al nombre de atributo. Por ejemplo, si una tabla tiene una columna 'for', el modelo generado tendrá un campo 'for_field', con el atributo db_column establecido en 'for'. inspectdb insertará el comentario Python 'Field renamed because it was a Python reserved word.' junto al campo.

Esta característica está pensada como un atajo, no como la generación de un modelo definitivo. Después de ejecutarla, querrás revisar los modelos generados para personalizarlos. En particular, necesitarás reordenar los modelos de manera tal que las relaciones estén ordenadas adecuadamente.

Las claves primarias son detectadas automáticamente durante la introspección para PostgreSQL, MySQL, y SQLite, en cuyo caso Django coloca primary_key=True donde sea necesario.
inspectdb trabaja con PostgreSQL, MySQL, y SQLite. La detección de claves foráneas solo funciona en PostgreSQL y con ciertos tipos de tablas MySQL.

La opción --database puede ser usada para especificar la base de datos a introspecciónar.

loaddata [fixture fixture ...]

django-admin.py loaddata

Busca y carga el contenido del 'fixture' nombrado en la base de datos.
La opción --database puede ser usada para especificar la base de datos sobre la cual cargar los datos.

--ignorenonexistent

La opción --ignorenonexistent puede ser usada para ignorar campos y modelos que hayan sido removidos desde que fixture fue generada originalmente

--app

La opción –app puede ser usada para especificar una sola aplicación donde buscar fixtures en lugar de buscar en todas las aplicaciones.

¿Qué es un "fixture"?

Un **fixture** es una colección de archivos que contienen los contenidos de la base de datos serializados. Cada fixture tiene un nombre único; de todas formas, los archivos que conforman el fixture pueden estar distribuidos en varios directorios y en varias aplicaciones.
Django buscará fixtures en tres ubicaciones:

1. En el directorio fixtures de cada aplicación instalada.

2. En todo directorio nombrado en la configuración FIXTURE_DIRS

3. En el path literal nombrado por el fixture

Django cargará todos los fixtures que encuentre en estas ubicaciones que coincidan con los nombres de fixture dados.

Si el fixture nombrado tiene una extensión de archivo, sólo se cargarán fixtures de ese tipo. Por ejemplo lo siguiente:

```
django-admin.py  loaddata mydata.json
```

Sólo cargará fixtures JSON llamados mydata. La extensión del fixture debe corresponder al nombre registrado de un serializador (ej.: json o xml).

Si omites la extensión, Django buscará todos los tipos de fixture disponibles para un fixture coincidente. Por ejemplo, lo siguiente:

```
django-admin.py loaddata mydata
```

Buscará todos los fixture de cualquier tipo de fixture llamado mydata. Si un directorio de fixture contiene mydata.json, ese fixture será cargado como un fixture JSON. De todas formas, si se descubren dos fixtures con el mismo nombre pero diferente tipo (ej.: si se encuentran mydata.json y mydata.xml en el mismo directorio de fixture), la instalación de fixture será abortada, y todo dato instalado en la llamada a loaddata será removido de la base de datos.

Los fixtures que son nombrados pueden incluir como componentes directorios. Estos directorios serán incluidos en la ruta de búsqueda. Por ejemplo, lo siguiente:

```
django-admin.py loaddata foo/bar/mydata.json
```

Buscará <appname>/fixtures/foo/bar/mydata.json para cada aplicación instalada, <dirname>/foo/bar/mydata.json para cada directorio en FIXTURE_DIRS, y la ruta literal foo/bar/mydata.json.

Observa que el orden en que cada fixture es procesado es indefinido. De todas formas, todos los datos de fixture son instalados en una única transacción, por lo que los datos en un fixture pueden referenciar datos en otro fixture. Si el back-end de la base de datos admite restricciones a nivel de registro, estas restricciones serán chequeadas al final de la transacción.

El comando dumpdata puede ser usado para generar la entrada para loaddata.

Comprimir fixtures

Los fixtures pueden ser comprimidos en formato zip, gz, o bz2. Por ejemplo:

```
django-admin.py  loaddata mydata.json
```

El comando anterior buscaría cualesquiera de: mydata.json, mydata.json.zip, mydata.json.gz, o mydata.json.bz2. El primer archivo que contenga dentro uno archivo comprimido ZIP será usado.

MYSQL y los fixtures

Desafortunadamente, MySQL no es capaz de dar soporte completo para todas las características de las fixtures de Django. Si usas tablas MyISAM, MySQL no admite transacciones ni restricciones, por lo que no tendrás rollback si se encuentran varios archivos de transacción, ni validación de los datos de fixture. Si usas tablas InnoDB,

no podrás tener referencias hacia adelante en tus archivos de datos – MySQL no provee un mecanismo para retrasar el chequeo de las restricciones de registro hasta que la transacción es realizada.

fixtures específicos en bases de datos

Si estas instalando múltiples bases de datos, tal vez tengas algunos datos que quieras cargar en una base de datos, pero no en otra. En esta situación puedes agregar el identificador de la base de datos en el nombre de los fixtures.

Por ejemplo, si la configuración DATABASES tiene una base de datos definida como maestra, el nombre del fixture puede ser datos.maestra.json o datos.maestra.json.gz, de esta forma el fixture únicamente cargara los datos en la base de datos llamada maestra.

makemessages

django-admin.py makemessages

Se ejecuta completamente sobre el directorio actual y recopila todas las cadenas marcadas para traducción. Crea (o actualiza) un archivo de mensajes en conf/locale (en el árbol de Django) o en el directorio local (para el proyecto y aplicación). Después de hacer los cambios a los archivos de mensajes, es necesario compilarlos con compilemessages para usarlos con el soporte incorporado gettext.

Usa la opción --all o -a para actualizar un archivo de mensajes para todos los lenguajes disponibles.

Ejemplo de su uso:

```
django-admin.py  makemessages –all
```

--extension

Usa la opción --extension o -e para especificar una lista de extensiones de archivos a examinar (default:”.html”, ”.txt”).

Ejemplo de su uso:

```
django-admin.py  makemessages --locale=de --extension xhtml
```

Separa múltiples extensiones con comas o usa -e o –extension varias veces:

```
django-admin.py  makemessages --locale=de --extension=html,txt --extension xml
```

- Usa la opción *--locale* (o su version corta -l) para especificar procesos de localización(es).
- Usa la opción *--exclude* (o su version corta``-x``) para especificar localización(es) para excluir del procesamiento. Si no se le provee, ninguna localización será excluida.

Ejemplo de su uso:

```
django-admin.py  makemessages --locale=pt_BR
django-admin.py  makemessages --locale=pt_BR --locale=fr
django-admin.py  makemessages -l pt_BR
django-admin.py  makemessages -l pt_BR -l fr
```

```
django-admin.py makemessages --exclude=pt_BR
django-admin.py makemessages --exclude=pt_BR --exclude=fr
django-admin.py makemessages -x pt_BR
django-admin.py makemessages -x pt_BR -x fr
```

makemigrations [<app_label>]

django-admin.py makemigrations

Crea las nuevas migraciones basadas en los cambios detectados en los modelos.

Proveer uno o más nombres de aplicaciones como argumentos limitará las migraciones creadas en las aplicaciones especificadas y cualquier dependencia necesaria (por ejemplo en las tablas del otro extremo de las relaciones ForeignKey)

--empty

La opción --empty permitirá que la salida de makemigrations sea una migración vacía para la aplicación especificada, para editarla manualmente. Esta opción debería ser usada, solo por usuarios avanzados, que estén familiarizados con el formato de migraciones, operaciones de migraciones y las dependencias entre migraciones.

--dry-run

La opción --dry-run muestra que migraciones serán aplicadas, sin escribir en los archivos de migraciones del disco. Usa esta opción con --verbosity 3 para mostrar los archivos completos de migraciones que serán escritos.

--merge

La opción --merge permite corregir conflictos de migraciones. La opción --noinput puede ser provista para suprimir mensajes de confirmación en la terminal del usuario, como confirmaciones de borrado, etc. durante la fusión.

--name -n

La opción --name permite dar a las migraciones un nombre personalizado en vez del generado automáticamente.

migrate [<app_label> [<migrationname>]]

django-admin.py migrate

Sincroniza el estado de la base de datos, con el actual conjunto de modelos y migraciones.

El comportamiento de este comando cambia dependiendo de los argumentos provistos:

- Sin argumentos: Todas las aplicaciones que contienen migraciones son migradas y todas las aplicaciones no emigradas son sincronizadas con la base de datos.

- <app_label>: La aplicación en particular que tiene migraciones a ejecutar, hasta la migración reciente. Esto puede implicar ejecutar las migraciones de otras aplicaciones también, debido a las dependencias.

- <app_label> <migrationname>: Trae el esquema de la base de datos a un estado donde se ejecuto la migración dada, pero no más allá – esto conlleva desaplicar las migraciones si se ha emigrado previamente a la denominado migración. Usa el nombre zero para desaplicar todas las migraciones para una aplicación.

--fake

La opción --database puede usarse para especificar la base de datos a migrar.

La opción --fake le dice a Django que marque la migración como aplicadas o no aplicadas, pero sin ejecutar SQl realmente, para cambiar el esquema de la base de datos.

Está diseñado para el uso de usuarios avanzados que pueden manipular directamente el estado actual de las migraciones, si están aplicando manualmente cambios; sea cuidadoso usando y ejecutando --fake, ya que se corre el riesgo de poner las tablas en un estado, donde será necesaria la recuperación manual, para que las migraciones se ejecutan correctamente.

--list, -l

La opción --list permite listar todas las aplicaciones que Django sabe que tienen disponibles migraciones por cada aplicación y si están aplicadas o no (marcándolas con una [X] junto al nombre de la migración)

Las aplicaciones sin migraciones son únicamente incluidas en la lista, pero con el nombre (no migrations) impreso debajo de ellas.

runserver [número de puerto opcional, o direcciónIP:puerto]

django-admin.py runserver

Inicia un servidor Web liviano de desarrollo en la máquina local. Por omisión, el servidor se ejecuta en el puerto 8000 de la dirección IP 127.0.0.1. Puedes pasarle explícitamente una dirección IP y un número de puerto.

Si ejecutas este script como un usuario con privilegios normales (recomendado), puedes no tener acceso a iniciar un puerto en un número de puerto bajo. Los números de puerto bajos son reservados para el superusuario (root).

☠ **Advertencia: No uses este servidor en una configuración de producción**. No se le han realizado auditorías de seguridad o tests de performance, y no hay planes de cambiar este hecho. Los desarrolladores de Django están en el negocio de hacer Web frameworks, no servidores Web, por lo que mejorar este servidor para que pueda manejar un entorno de producción está fuera del alcance de Django.

El servidor de desarrollo carga automáticamente el código Python para cada pedido según sea necesario. No necesitas reiniciar el servidor para que los cambios en el código tengan efecto.

Cuando inicias el servidor, y cada vez que cambies código Python mientras el servidor está ejecutando, éste validará todos tus modelos instalados. (Ver la sección

que viene sobre el comando validate.) Si el validador encuentra errores, los imprimirá en la salida estándar, pero no detendrá el servidor.

Puedes ejecutar tantos servidores como quieras, siempre que ejecuten en puertos separados. Sólo ejecuta django-admin.py runserver más de una vez.

Observa que la dirección IP por omisión, 127.0.0.1, no es accesible desde las otras máquinas de la red. Para hacer que el servidor de desarrollo sea visible a las otras máquinas de la red, usa su propia dirección IP (ej.: 192.168.2.1) o 0.0.0.0.

Por ejemplo, para ejecutar el servidor en el puerto 7000 en la dirección IP 127.0.0.1, usa esto:

```
django-admin.py runserver 9000
```

O para ejecutar el servidor en el puerto 9000 en la dirección IP 1.2.3.4, usa esto:

```
django-admin.py runserver 1.2.3.4:9000
```

--noreload

Usa la opción --noreload para deshabilitar el uso de él recargado automático. Esto significa que cualquiera de los cambios que hagas en el codigo, no hara que el servidor se recargue, mientras el servidor se está ejecutando, por lo que solo se usaran los módulos específicos de Python que se hayan cargado en la memoria.

Ejemplo de su uso:

```
django-admin.py  runserver –noreload
```

--ipv6, -6

Usa la opción --ipv6 (o su version corta -6) para decirle a Django que use IPv6 para el servidor de desarrollo. Esta cambia la dirección predeterminada IP de 127.0.0.1 a ::1.

Ejemplo de su uso:

```
django-admin.py  runserver --ipv6
```

Ejemplos de diferentes usos de puertos y direcciones

Puerto 8000 en dirección IP 127.0.0.1:

```
django-admin.py  runserver
```

Puerto 8000 en dirección IP 1.2.3.4:

```
django-admin.py  runserver 1.2.3.4:8000
```

Puerto 7000 en dirección IP 127.0.0.1:

```
django-admin.py  runserver 7000
```

Puerto 7000 en dirección IP 1.2.3.4:

```
django-admin.py  runserver 1.2.3.4:7000
```

Puerto 8000 en dirección IPv6 ::1:

```
django-admin.py runserver -6
```

Puerto 7000 en dirección IPv6 ::1:

```
django-admin.py runserver -6 7000
```

Puerto 7000 en dirección IPv6 2001:0db8:1234:5678::9:

```
django-admin.py runserver [2001:0db8:1234:5678::9]:7000
```

Puerto 8000 en dirección IPv4 del host localhost:

```
django-admin.py runserver localhost:8000
```

Puerto 8000 en la dirección IPv6 del host localhost::

```
django-admin.py runserver -6 localhost:8000
```

shell

django-admin.py shell

Inicia el intérprete interactivo de Python.
Django utilizará IPython o bpython si están instalados, para iniciar un shell mejorado. Pero si quieres forzar el uso del intérprete Python "plano", usa la opción --plain, como en:

```
django-admin.py shell –plain
```

Si quieres especificar entre IPython o bpython como tu interprete, si tienes ambos instalados, puedes especificar alternativamente la interface que quieras usar, usando la opción -i o --interface así:

IPython:

```
django-admin.py shell -i ipython
```

```
django-admin.py shell --interface ipython
```

bpython:

```
django-admin.py shell -i bpython
```

```
django-admin.py shell --interface bpython
```

Cuando el interprete interactivo "plano" inicia,(ya sea porque usaste la opción --plain o porque no tienes otra interface disponible) este lee el script que apunta a las variables de entorno de PYTHONSTARTUP y al script de la variable ~/.pythonrc.py , si quieres cambiar este comportamiento usa la opción --no-startup, por ejemplo:

sql <app_label app_label ...>

django-admin.py sql

Imprime las sentencias SQL CREATE TABLE para las aplicaciones mencionadas.

La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqlall <app_label app_label ...>

django-admin.py sqlall

Imprime las sentencias SQL CREATE TABLE y los datos iniciales para las aplicaciones mencionadas.

Busca en la descripción de sqlcustom para una explicación de cómo especificar los datos iniciales.

La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqlclear <app_label app_label ...>

django-admin.py sqlclear

Imprime las sentencias SQL DROP TABLE para las aplicaciones mencionadas.

La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqlcustom <app_label app_label ...>

django-admin.py sqlcustom

Imprime las sentencias SQL personalizadas para las aplicaciones mencionadas.

Para cada modelo en cada aplicación especificada, este comando busca el archivo <appname>/sql/<modelname>.sql, donde <appname> es el nombre de la aplicación dada y <modelname> es el nombre del modelo en minúsculas. Por ejemplo, si tienes una aplicación news que incluye un modelo Story, sqlcustom tratará de leer un archivo news/sql/story.sql y lo agregará a la salida de este comando.

Se espera que cada uno de los archivos SQL, si son dados, contengan SQL válido. Los archivos SQL son canalizados directamente a la base de datos después que se hayan ejecutado todas las sentencias de creación de tablas de los modelos. Usa este enlace SQL para hacer cualquier modificación de tablas, o insertar funciones SQL en las bases de datos.

Observa que el orden en que se procesan los archivos SQL es indefinido. La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqldropindexes <app_label app_label ...>

django-admin.py sqldropindexes

Imprime las sentencias SQL DROP INDEX SQL para las aplicaciones mencionadas.

La opción –database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqlflush

django-admin.py sqlflush

Imprime las sentencias SQL que serán ejecutadas por el comando flush.

La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqlindexes <app_label app_label ...>

django-admin.py sqlindexes

Imprime las sentencias SQL CREATE INDEX para las aplicaciones mencionadas.
La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

sqlmigrate <app_label> <migrationname>

django-admin.py sqlmigrate

Imprime el SQL para el nombre de la migración. Esta requiere una conexión a una base de datos activa, que se utilizara para resolver restricciones de nombres; esto significa que se debe generar el SQL contra una copia de la base de datos que se desea aplicar más adelante.
Observa que sqlmigrate no coloriza la salida.
La opción –database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

--backwards

De forma predeterminada el SQL creado, permite ejecuta las migraciones que siguen. Pasándole la opción --backwards al SQL generado permite desaplicar las migraciones.

sqlsequencereset <app_label app_label ...>

django-admin.py sqlsequencereset

Imprime las sentencias SQL para resetear series para las aplicaciones mencionadas. Las series o secuencias son índices usados por algunos motores de base de datos para rastrear el número disponible siguiente para los campos automáticamente incrementados.
Use este comando para generar SQL que corrija casos donde una secuencia está fuera de sincronización con campos automáticamente incrementados.
La opción --database puede ser usada para especificar la base de datos sobre la cual imprimir el SQL.

squashmigrations <app_label> <migration_name>

django-admin.py squashmigrations

Compacta las migraciones para una aplicación dada (app_label), hasta el nombre de la migración (migration_name) incluyéndola y compactándolas lo más posible, las migraciones resultantes pueden convivir de forma segura con migraciones `no compactadas (unsquashed).

--no-optimize

De forma predeterminada, Django trata de optimizar las operaciones de migraciones para reducir el tamaño de los archivos resultantes. Sin embargo puedes usar la opción --no-optimize, si el proceso está fallando o si está creando migraciones incorrectas.

startapp <app_label> [destino]

django-admin.py startapp

Crea una estructura de directorios para una aplicación Django con el nombre de la aplicación dada, en el directorio actual.

De forma predeterminada el directorio creado contiene un archivo models.py y otros archivos para una aplicación (`views.py, admin.py...).

Si se le proporciona un destino opcional, Django utilizará el directorio existente para crear uno nuevo. Puedes utilizar".'' para denotar el directorio de trabajo actual.

Por ejemplo:

```
django-admin.py startapp myapp /Users/jezdez/Code/myapp
```

--template

Con la opción template --template, puedes utilizar plantillas personalizadas para una aplicación, proporcionando la ruta al directorio que contiene las plantillas de archivos o la ruta a los archivos comprimidos (.tar.gz, .tar.bz2, .tgz, .tbz, .zip) que puedan contener los archivos de plantillas para la aplicación.

Por ejemplo, esto buscaría plantillas para una aplicación en el directorio miapp:

```
django-admin.py startapp --template=/Users/jezdez/Code/my_app_template myapp
```

Django también acepta URLs (http, https, ftp) para archivos comprimidos que contengan archivos de plantillas para una aplicación, descargando y extrayendo los archivos al vuelo.
Por ejemplo, aprovechándo la característica de Github de exponer repositorios como archivos zip, puedes utilizar un URL como esta:

```
django-admin.py startapp --template=https://github.com/githubuser/django-app-
template/archive/master.zip myapp
```

Cuando Django copia los archivos de plantillas de una aplicación, también renderiza ciertos archivos a través del motor de plantillas: los archivos cuyas extensiones coincidan con la opción --extension (py por defecto) y los archivos cuyos nombres sean pasados con la opción --name.

La clase django.template.Context usa para esto, cualquier opción pasada al comando startapp (Entre las opciones soportadas por el comando)

- *app_name* – el nombre de la aplicación pasada al comando.
- *app_directory* – la ruta completa a la aplicación recién creada.
- *docs_version* – la versión de la documentación: 'dev' o '1.x'

startproject <projectname> [destino]

django-admin.py startproject

Crea una estructura de directorios Django para el nombre de proyecto dado, en el directorio actual o en el directorio dado.

De forma predeterminada, el nuevo directorio contiene un archivo manage.py y un paquete de proyecto (que contiene un archivo settings.py y otros archivos), si sólo el nombre de proyecto es dado, tanto el directorio del proyecto y el paquete del proyecto serán llamados <projectname> y el directorio del proyecto será creado en el directorio de trabajo actual.

Si se le provee la opción destino, Django usara el directorio existente como el directorio del proyecto y creara un archivo manage.py y el paquete del proyecto dentro del. Usa '.' para denotar el actual directorio de trabajo.

Por ejemplo:

```
django-admin.py startproject myproject /Users/jezdez/Code/myproject_repo
```

Al igual que el comando startapp la opción --template permite especificar un directorio, una ruta de archivos o una URL para proveer de plantillas personalizadas al proyecto.

Por ejemplo, esto buscara plantillas para un proyecto en el directorio dado, cuando se cree el proyecto myproject:

```
django-admin.py startproject --template=/Users/jezdez/Code/my_project_template
  myproject
```

También acepta URLs (http, https, ftp) para archivos comprimidos que contengan archivos de plantillas para un proyecto, descargando y extrayendo los archivos al vuelo.

Por ejemplo, aprovechándo la característica de Github de exponer repositorios como archivos zip, puedes utilizar un URL como esta:

```
django-admin.py startproject --template=https://github.com/githubuser/django-project-
  template/archive/master.zip myproject
```

Cuando Django copia los archivos de plantillas de un proyecto, también renderiza ciertos archivos a través del motor de plantillas: los archivos cuyas extensiones coincidan con la opción --extension (py por defecto) y los archivos cuyos nombres sean pasados con la opción --name.

La clase django.template.Context usa para esto, cualquier opción pasada al comando startapp (Entre las opciones soportadas por el comando)

- project_name – el nombre del proyecto pasado al comando.
- project_directory – la ruta completa al proyecto recién creado.
- secret_key – una clave al azar para la configuración de SECRET_KEY.
- docs_version – la versión de la documentación: 'dev' o '1.x'

test

django-admin.py test

Ejecuta todas las pruebas para todos los modelos instalados.

--failfast

La opción --failfast se usa para detener las pruebas y reportar las fallas inmediatamente después de que una prueba falla.

--testrunner

La opción --testrunner es usada para controlar la clase de pruebas que es usada para ejecutar las pruebas. Si este valor es proporcionado al comando, sobrescribirá el valor provisto por la configuración TEST_RUNNER.

--liveserver

La opción --liveserver es usada para sobrescribir la dirección predeterminada que usa el servidor de pruebas, (usado con LiveServerTestCase.) El valor predeterminado es localhost:8081.

--keepdb

La opción --keepdb es usada para preservar los test en la base de datos entre la ejecución de pruebas. Este toma la ventaja de saltarse el crear y destruir las tablas entre prueba y prueba, esto disminuye en gran medida el tiempo de pruebas, especialmente en grandes suits de pruebas. Si la base de datos de tests no existe, esta se crea la primera vez que se ejecuta y se preserva en cada subsecuente ejecución. Cualquier migración no aplicada será aplicada a la base de datos de pruebas antes de ejecutar todas las pruebas.

testserver <fixture fixture ...>

django-admin.py testserver

Ejecuta un servidor de desarrollo Django (como runserver) usando datos proporcionados por fixture(s)

Por ejemplo, este comando:

```
django-admin.py testserver mydata.json
```

Realizara los siguientes pasos:

1. Creara una base de datos de pruebas.
2. Poblara la base de datos de pruebas con datos provistos por los fixtures.
3. Ejecutar el servidor de desarrollo (tal como runserver) apuntado a la base de datos recién creada en lugar de la base de datos de producción.

Comandos provistos por aplicaciones

Algunos comando son únicamente disponibles cuando la aplicación django.contrib ha sido activada. Esta sección describe los comandos provistos por cada aplicación.

django.contrib.auth

changepassword

django-admin.py changepassword

Este comando se encuentra únicamente disponible, si se instala el sistema de autentificación (django.contrib.auth.)

Permite cambiar las contraseñas de los usuarios. Instiga a entrar dos veces la contraseña de un usuario dado, como parámetro. Si ambos coinciden, la nueva

contraseña será cambiada de inmediato. Si no se le pasa un usuario, el comando intentara usar el usuario actual para cambiar la contraseña.

Usa la opción --database para especificar la base de datos a consultar para el usuario. Si esta no se le proporciona, Django usara la base de datos por default

Ejemplo de sus uso:

```
django-admin.py changepassword ringo
```

createsuperuser

django-admin.py createsuperuser

Este comando se encuentra únicamente disponible, si se instala el sistema de autentificación (django.contrib.auth.)

Crea una cuenta de superusuario (un usuario que tiene todos los permisos). Esto es útil si necesitas crear una cuenta inicial para un superusuario o si necesitas generar programáticamente cuentas de superusuarios para tu sitio(s).

Cuando se ejecuta interactivamente, este comando preguntara por una contraseña para crear la cuenta del superusuario. Cuando se ejecuta de forma no interactiva, es necesario configurar la contraseña de forma manual.

--username
--email

El nombre de usuario y la dirección de email para la nueva cuenta, pueden proporcionarse usando los argumentos --username y --email en la línea de comandos. Si cualquiera de ellos no se provee, createsuperuser preguntara por ellos al ejecutarse de forma interactiva.

Usa la opción --database para especificar la base de datos en la que quieras guardar los objetos superusuarios.

django.contrib.gis

ogrinspect

Este comando se encuentra únicamente disponible, si se instala el sistema de Geodjango(django.contrib.gis).

django.contrib.sessions

clearsessions

django-admin.py clearsessions

Puede ser usado como una tarea por cron o directamente para limpiar las sesiones que han expirado.

django.contrib.sitemaps

ping_google

Este comando se encuentra únicamente disponible, si se instala el framework de Sitemaps (django.contrib.sitemaps).

Usado para hacer ping a Google, para que indexe nuestro sitio(s). Una vez que has agregado la aplicación sitemap a tu proyecto, puedes hacer ping a Google usando este comando de la siguiente forma:

```
python manage.py ping_google [/sitemap.xml]
```

django.contrib.staticfiles

collectstatic

Este comando se encuentra únicamente disponible, si se instala el framework de archivos estáticos (django.contrib.staticfiles).

Usado para coleccionar los archivos estáticos (hojas de estilo css, js...) que serán servidos por el servidor web en un entorno de producción. Los nombres duplicados de los archivos de forma predeterminada se resuelven de una manera similar a como lo hacen las plantillas: el archivo que es primero localizado, es el que se usara.

Colecciona los archivos estáticos en STATIC_ROOT. Para una lista completa de opciones, usa --help:

```
python manage.py collectstatic –help
```

findstatic

Búsca uno o más caminos relativos, a la ruta de los buscadores habilitados.

Por ejemplo:

```
python manage.py findstatic css/base.css admin/js/core.js
```

De forma predeterminada, todas las localidades que coincidan serán encontradas. Si únicamente quieres que devuelva la primera coincidencia por cada ruta relativa, usa la opción --first así:

```
python manage.py findstatic css/base.css –first
```

Opciones Predeterminadas

Aunque algunos comandos pueden permitir sus propias opciones personalizadas, cada comando permite usar las siguientes opciones:
Las secciones que siguen delinean las opciones que puede tomar django-admin.py en la mayoría de casos.

--settings

Especifica explícitamente el módulo de configuración a usar. El módulo de configuración debe estar en la sintaxis de paquetes de Python (ej.: mysite.settings). Si no se proveen, django-admin.py utilizará la variable de entorno DJANGO_SETTINGS_MODULE.

Ejemplo de uso:

```
django-admin.py migrate --settings=mysite.settings
```

Observa que esta opción no es necesaria en manage.py, ya que toma en cuenta la configuración de DJANGO_SETTINGS_MODULE por tí.

--pythonpath

Agrega la ruta del sistema de archivos a la ruta de búsqueda de importación de Python. Si no se define, django-admin.py usará la variable de entorno PYTHONPATH.
Ejemplo de uso:

```
django-admin.py migrate --pythonpath='/home/djangoprojects/myproject'
```

Observa que esta opción no es necesaria en manage.py, ya que este comando tiene cuidado de configurar la ruta de Python por tí.

--format

Especifica el formato de salida que será utilizado. El nombre provisto debe ser el nombre de un serializador registrado.

Ejemplo de uso:

```
django-admin.py dumpdata --format=xml
```

--help

Muestra un mensaje de ayuda que incluye una larga lista de todas las opciones y acciones disponibles.

–traceback

De forma predeterminada django-admin muestra un simple mensaje de error cuando ocurre un CommandError, pero muestra una traza completa para cualquier otra excepción. Si especificas la opción --traceback, django-admin también mostrara la traza de pila completa cuando ocurra algún error de comando o CommandError.
Por ejemplo:

```
django-admin.py migrate –traceback
```

--no-color

De forma predeterminada el formato que usa django-admin para mostrar las salidas es colorizado. Por ejemplo, los errores se imprimen en la consola en rojo y las declaraciones en SQL aparecen con resaltado de sintaxis. Para prevenir esto y mostrar la salida de texto plano, es necesario pasarle la opción --no-color al ejecutar el comando, de la siguiente forma:

```
django-admin.py sqlall --no-color
```

--version

Muestra la versión actual de Django.

```
django-admin.py version
```

Ejemplo de salida:

```
1.8
1.9
```

--verbosity

Determina la cantidad de notificaciones e información de depuración que se imprimirá en la consola.

Ejemplo de uso:

```
django-admin.py migrate --verbosity=2
```

Usa niveles para determinar la cantidad de información a mostrar:

0 significa sin salida.
1 significa salida normal (default).
2 significa salida con explicaciones.
3 significa *muy* explicado.

Opciones comunes

Las siguientes opciones no están disponibles en cada comando, pero son muy comunes en un gran número de comandos.

–database

Usado para especificar la base de datos en la cual opera el comando. Si no se especifica, esta opción usara el valor predeterminado por el alias default.

Por ejemplo, para volcar datos de una base de datos con el alias master:

```
django-admin.py dumpdata --database=master
```

–exclude

Excluye una aplicación en específico de las aplicaciones cuyo contenido se espera en la salida.

Por ejemplo, para específicamente excluir la aplicación auth de la salida de dumpdata. Por ejemplo:

```
django-admin.py dumpdata --exclude=auth
```

Si quieres excluir múltiples aplicaciones, usa múltiples directivas --exclude:

```
django-admin.py dumpdata --exclude=auth --exclude=contenttypes
```

–indent

Especifica el número de espacios que se utilizarán para la indentación cuando se imprima una salida con formato de impresión. Por omisión, la salida *no* tendrá formato de impresión. El formato de impresión solo estará habilitado si se provee la opción de indentación.

Ejemplo de uso:

```
django-admin.py dumpdata --indent=4
```

--noinput

Indica que no quieres que se te pida ninguna entrada. Es útil cuando el script django-admin se ejecutará en forma automática y desatendida.

–noreload

Deshabilita el uso del autoreloader cuando se ejecuta el servidor de desarrollo.

–locale

Usa la opción --locale o -l para específicamente procesar una localidad. Si no provees esta opción todas las localidades serán procesadas.

Sutilezas extras

Sintaxis de colores

Los comandos django-admin / manage.py usan un agradable resaltado de código en la salida (colores) mostrada en la terminales con soporte para salida ANSI-colored.

Bajo Windows, la consola nativa no soporta el escape de secuencias ANSI, por lo que la salida no es mostrada con colores. Pero puedes instalar una herramienta de terceros llamada ANSICON (http:adoxa.altervista.org/ansicon/), el comando de Django puede detectar si está presente y usarla para mostrar el resaltado de sintaxis, tal como en plataformas basadas en Unix.

Los colores usados por el resaltado de sintaxis pueden personalizarse. Django viene con tres paletas de colores.

- **dark**, adaptado apara terminales que muestran texto en blanco y el fondo en negro. Esta es la paleta predeterminada.
- **light**, adaptado para terminales que muestran el texto en negro en un fondo blanco.
- **nocolor**, desactiva el resaltado de sintaxis.

Puedes seleccionar una paleta configurando las variables de entorno para DJANGO_COLORS especificando la paleta a usar. Por ejemplo, para especificar la paleta light bajo un shell tipo Unix o OS/X tipo shell BASH, puedes usar el siguiente comando en una terminal:

```
export DJANGO_COLORS="light"
```

Autocompletado para bash

Si estas usando el shell Bash, considera instalar el script bash completion el cual se localiza en extras/django_bash_completion en la distribución de Django. Este habilita el autocompletado usando la tecla tab, para los comandos django-admin.py y manage.py, con lo que es posible, por ejemplo...

Tipea `django-admin.py`:

- Presionar [TAB] para ver las opciones disponibles.
- Tipea sql, luego [TAB], para ver todas las opciones disponibles cuyos nombres comienzan con sql.

APÉNDICE G

Objetos petición y respuesta

Usa los objetos petición y respuesta para pasar información de estado a través del sistema.

Cuando se peticiona una página, Django crea un objeto HttpRequest que contiene metadatos sobre la petición. Luego Django carga la vista apropiada, pasando el HttpRequest como el primer argumento de la función de vista. Cada vista es responsable de retornar un objeto HttpResponse.

Hemos usado estos objetos con frecuencia a lo largo del libro; este apéndice explica las APIs completas para los objetos HttpRequest (petición) y HttpResponse (respuesta).

HttpRequest

HttpRequest representa una sola petición HTTP desde algún agente de usuario.

Mucha de la información importante sobre la petición está disponible como atributos en la instancia misma de HttpRequest (mira la Tabla G-1). Todos los atributos excepto **session** deben considerarse de sólo lectura.

Atributo	Descripción
path	Un string que representa la ruta completa a la página peticionada, no incluye el dominio – por ejemplo, "/music/bands/the_beatles/".

method	Un string que representa el método HTTP usado en la petición. Se garantiza que estará en mayúsculas. Por ejemplo:
	
	`if request.method == 'GET':`
	`  do_something()`
	`elif request.method == 'POST':`
	`  do_something_else()`

encoding	Una cadena que representa la actual codificación usada para decodificar los datos enviados por los formularios (o None), lo cual significa que DEFAULT_CHARSET se está usando. Puedes rescribir este atributo para cambiar la codificación usada para acceder a los datos del formulario. Cualquier subsecuente acceso a los atributos (tales como leer de GET o POST) usaran el nuevo valor del encoding. Útil si sabes que los datos del formulario no están en la en codificación de DEFAULT_CHARSET.
GET	Un objeto similar a un diccionario que contiene todos los parámetros HTTP GET dados. Mira la documentación de *QueryDict* ` que sigue.

POST	Un objeto similar a un diccionario que contiene todos los parámetros HTTP POST dados.Mira la documentación de QueryDict que sigue. Es posible que una petición pueda ingresar vía POST con un diccionario POST vacío – si, digamos, un formulario es peticionado a través del método HTTP POST pero que no incluye datos de formulario. Por eso, no deberías usar if request.POST para verificar el uso del método POST; en su lugar, utiliza if request.method == "POST" (mira la entrada method en esta tabla). Nota: POST *no* incluye información sobre la subida de archivos. Mira FILES.
REQUEST	Por conveniencia, un objeto similar a un diccionario que busca en POST primero, y luego en GET. Inspirado por $_REQUEST de PHP. Por ejemplo, si GET = {"name": "john"} y POST = {"age": '34'}, REQUEST["name"] será "john", y REQUEST["age"] será "34". Se sugiere encarecidamente que uses GET y POST en lugar de REQUEST, ya que lo primero es más explícito.
COOKIES	Un diccionario Python estándar que contiene todas las cookies. Las claves y los valores son strings. Mira el *Capítulo 12* para saber más de cookies.
FILES	Un objeto similar a un diccionario que contiene todos los archivos subidos. Cada clave de FILES es el atributo name de <input type="file" name="" />. Cada valor de FILES es un diccionario Python estándar con las siguientes tres claves: ▪ *filename*: El nombre del archivo subido, como un string Python. ▪ *content-type*: El tipo de contenido del archivo subido. ▪ *content:* El contenido en crudo del archivo subido. Nota que FILES contendrá datos sólo si el método de la petición fue POST y el <form> que realizó la petición contenía enctype="multipart/form-data". De lo contrario, FILES será un objeto similar a un diccionario vacío.
META	Un diccionario Python estándar que contiene todos los encabezados HTTP disponibles. Los encabezados disponibles dependen del cliente y del servidor, pero estos son algunos ejemplos: ▪ CONTENT_LENGTH ▪ CONTENT_TYPE ▪ QUERY_STRING: La string de consulta en crudo sin analizar. ▪ REMOTE_ADDR: La dirección IP del cliente. ▪ REMOTE_HOST: El nombre host del cliente. ▪ SERVER_NAME: El nombre host del servidor. ▪ SERVER_PORT: El puerto del servidor. Cualquier cabecera HTTP está disponible en META como claves con el prefijo HTTP_, por ejemplo: ▪ HTTP_ACCEPT_ENCODING ▪ HTTP_ACCEPT_LANGUAGE

	▪ HTTP_HOST: La cabecera HTTP host en viada por el cliente ▪ HTTP_REFERER: La pagina referente, si la hay ▪ HTTP_USER_AGENT: La string de agente de usuario del cliente ▪ HTTP_X_BENDER: El valor de la cabecera X-Bender, si está establecida.

user	Un objeto django.contrib.auth.models.User que representa el usuario actual registrado. Si el usuario no está actualmente registrado, user se fijará a una instancia de django.contrib.auth.models.AnonymousUser. Puedes distinguirlos con is_authenticated(), de este modo:
	
	`if request.user.is_authenticated():`
	`  # Hacer algo para usuarios autentificados.`
	`else:`
	`# Hacer algo para usuarios anónimos.`
	
	user está disponible sólo si tu instalación Django tiene activado AuthenticationMiddleware. Para los detalles completos sobre autenticación y usuarios, mira el capítulo 14.

session	Un objeto similar a un diccionario que se puede leer y modificar, que representa la sesión actual. Éste está disponible sólo si tu instalación Django tiene activado el soporte para sesiones. Mira el capítulo 14.
raw_post_data	Los datos HTTP POST en crudo. Esto es útil para procesamiento avanzado.

Tabla G1: *Atributos de los objetos HttpRequest*

Los objetos request también tienen algunos métodos de utilidad, como se muestra en la Tabla G-2.

Método	Descripción
__getitem__(key)	Retorna el valor GET/POST para la clave dada, verificando POST primero, y luego GET. Emite KeyError si la clave no existe. Esto te permite usar sintaxis de acceso a diccionarios en una instancia HttpRequest. Por ejemplo, request["foo"] es lo mismo que comprobar request.POST["foo"] y luego request.GET["foo"].
has_key()	Retorna True o False, señalando si request.GET o request.POST contiene la clave dada.
get_host()	Devuelve el origen de la petición usando información de las cabeceras HTTP_X_FORWARDED_HOST y HTTP_HOST (en ese orden) Si no se le proveen valores, este método usa una combinación de SERVER_NAME y SERVER_PORT
get_full_path()	Retorna la ruta, más un string de consulta agregado. Por ejemplo, "/music/bands/the_beatles/?print=true"
is_secure()	Retorna True si la petición es segura; es decir si fue realizada con HTTPS.

Tabla G2: *Métodos de HttpRequest*

Objetos QueryDict

En un objeto HttpRequest, los atributos GET y POST son instancias de django.http.QueryDict. QueryDict es una clase similar a un diccionario personalizada para tratar múltiples valores con la misma clave. Esto es necesario ya que algunos elementos de un formulario HTML, en particular <select multiple="multiple">, pasan múltiples valores para la misma clave.

Las instancias QueryDict son inmutables, a menos que realices una copia de ellas. Esto significa que tú no puedes cambiar directamente los atributos de request.POST y request.GET.

QueryDict implementa todos los métodos estándar de los diccionarios, debido a que es una subclase de diccionario. Las excepciones se resumen en la Tabla G-3.

Método	Diferencias con la implementación estándar de dict
__getitem__	Funciona como en un diccionario. Sin embargo, si la clave tiene más de un valor, __getitem__() retorna el último valor.
__setitem__	Establece la clave dada a [value] (una lista de Python cuyo único elemento es value). Nota que ésta, como otras funciones de diccionario que tienen efectos secundarios, sólo puede ser llamada en un QueryDict mutable (uno que fue creado vía copy()).
get()	Si la clave tiene más de un valor, get() retorna el último valor al igual que __getitem__.

update()	Recibe ya sea un QueryDict o un diccionario estándar. A diferencia del método update de los diccionarios estándar, este método *agrega* elementos al diccionario actual en vez de reemplazarlos:
	
	>>> q = QueryDict('a=1')
	>>> q = q.copy() # to make it mutable
	>>> q.update({'a': '2'})
	>>> q.getlist('a')
	['1', '2']
	>>> q['a'] # returns the last
	['2']

items()	Similar al método items() de un diccionario estándar, excepto que éste utiliza la misma lógica del último-valor de __getitem()__:
	>>> q = QueryDict('a=1&a=2&a=3')
	>>> q.items()
	[('a', '3')]

values()	Similar al método values() de un diccionario estándar, excepto que este utiliza la misma lógica del último-valor de __getitem()__.

Tabla G3: *Como se diferencian los QueryDicts de los diccionarios estándar.*

Además, QueryDict posee los métodos que se muestran en la Tabla G-4.

Método	Descripción
copy()	Retorna una copia del objeto, utilizando copy.deepcopy() de la biblioteca estándar de Python. La copia será mutable – es decir, puedes cambiar sus valores.
getlist(key)	Retorna los datos de la clave requerida, como una lista de Python. Retorna una lista vacía si la clave no existe. Se garantiza que retornará una lista de algún tipo.
setlist(key, list_)	Establece la clave dada a list_ (a diferencia de __setitem__()).
appendlist(key, item)	Agrega un elemento item a la lista interna asociada a key.
setlistdefault(key, l)	Igual a setdefault, excepto que toma una lista de valores en vez de un sólo valor.

lists()	Similar a items(), excepto que incluye todos los valores, como una lista, para cada miembro del diccionario. Por ejemplo:
	
	>>> q = QueryDict('a=1&a=2&a=3')
	>>> q.lists()
	[('a', ['1', '2', '3'])]

urlencode()	Retorna un string de los datos en formato query-string (ej., "a=2&b=3&b=5").

Tabla G4 *Métodos QueryDict Extra (No relacionados con diccionarios)*

Un ejemplo completo

Por ejemplo, dado este formulario HTML:

```
<form action="/foo/bar/" method="post">
<input type="text" name="tu_nombre" />
<select multiple="multiple" name="bandas">
   <option value="beatles">The Beatles</option>
   <option value="who">The Who</option>
   <option value="zombies">The Zombies</option>
</select>
<input type="submit" />
</form>
```

Si el usuario ingresa "John Smith" en el campo "tu_nombre" y selecciona tanto “The Beatles” como “The Zombies” en la caja de selección múltiple, lo siguiente es lo que contendrá el objeto request de Django:

```
>>> request.GET
{}
>>> request.POST
{'tu_nombre': ['John Smith'], 'bandas': ['beatles', 'zombies']}
```

```
>>> request.POST[' tu_nombre']
'John Smith'
>>> request.POST['bandas']
'zombies'
>>> request.POST.getlist('bandas')
['beatles', 'zombies']
>>> request.POST.get('tu_nombre', 'Adrian')
'John Smith'
>>> request.POST.get('nonexistent_field', 'Nowhere Man')
'Nowhere Man'
```

■**Nota:**Los atributos GET, POST, COOKIES, FILES, META, REQUEST, raw_post_data, y user son todos cargados tardíamente. Esto significa que Django no gasta recursos calculando los valores de estos atributos hasta que tu código los solicita.

HttpResponse

A diferencia de los objetos HttpRequest, los cuales son creados automáticamente por Django, los objetos HttpResponse son tu responsabilidad. Cada vista que escribas es responsable de instanciar, poblar, y retornar un HttpResponse.

La clase HttpResponse está ubicada en django.http.HttpResponse.

Construcción de HttpResponses

Típicamente, tu construirás un HttpResponse para pasar los contenidos de la pagina, como un string, al constructor de HttpResponse:

```
>>> response = HttpResponse("Este es texto de una página Web.")
>>> response = HttpResponse("Únicamente texto.", mimetype="text/plain")
```

Pero si quieres agregar contenido de manera incremental, puedes usar response como un objeto similar a un archivo:

```
>>> response = HttpResponse()
>>> response.write("<p> Este es texto de una página Web.</p>")
>>> response.write("<p>Este es un parrafo.</p>")
```

Puedes pasarle a HttpResponse un iterador en vez de pasarle strings codificadas a mano. Si utilizas esta técnica, sigue estas instrucciones:

- El iterador debe retornar cadenas o strings.
- Si un HttpResponse ha sido inicializado con un iterador como su contenido, no puedes usar la instancia HttpResponse como un objeto similar a un archivo. Si lo haces, emitirá Exception.

Finalmente, nota que HttpResponse implementa un método write(), lo cual lo hace apto para usarlo en cualquier lugar que Python espere un objeto similar a un archivo. Mira él *Capítulo 11* para ver algunos ejemplos de la utilización de esta técnica.

Establecer las cabeceras

Puedes agregar o eliminar cabeceras usando sintaxis de diccionario:

```
>>> response = HttpResponse()
>>> response['X-DJANGO'] = "Es el mejor."
>>> del response['X-PHP']
>>> response['X-DJANGO']
" Es el mejor."
```

Puedes utilizar también has_header(header) para verificar la existencia de una cabecera.
Evita configurar cabeceras Cookie a mano; en cambio, mira él *capítulo 15* para instrucciones sobre cómo trabajan las cookies en Django.

Subclases de HttpResponse

Django incluye un número de subclases HttpResponse que manejan diferentes tipos de respuestas HTTP (mira la Tabla G-5). Así como HttpResponse, estas subclases se encuentran en django.http.

Clase	Descripción
HttpResponseRedirect	El constructor toma un único argumento: la ruta a la cual re-dirigir. Esta puede ser una URL completa (ej., 'http://search.yahoo.com/') o o una URL absoluta sin dominio (ej., '/search/'). Ten en cuenta que esto retorna un código de estado HTTP 302.
HttpResponsePermanentRedirect	Como HttpResponseRedirect, pero esta retorna una re-dirección permanente (código de estado HTTP 301) en vez de una re-dirección "found" (código de estado 302).
HttpResponseNotModified	El constructor no tiene ningún argumento. Utiliza esta para designar que una página no ha sido modificada desde la última petición del usuario.
HttpResponseBadRequest	Actúa como HttpResponse pero usa un código de estado 400.
HttpResponseNotFound	Actúa como HttpResponse pero usa un código de estado 404.
HttpResponseForbidden	Actúa como HttpResponse pero usa un código de estado 403.
HttpResponseNotAllowed	Como HttpResponse, pero usa un código de estado 405. Toma un único argumento: una lista de los métodos permitidos (ej., ['GET', 'POST']).

HttpResponseGone	Actúa como HttpResponse pero usa un código de estado 410.
HttpResponseServerError	Actúa como HttpResponse pero usa un código de estado 500

Tabla G5: *Subclasses de HttpResponse*

Puedes, por supuesto, definir tus propias subclases de HttpResponse para permitir diferentes tipos de respuestas no admitidas por las clases estándar.

Retornar Errores

Retornar códigos de error HTTP en Django es fácil. Ya hemos mencionado las subclases HttpResponseNotFound, HttpResponseForbidden, HttpResponseServerError, y otras. Simplemente retorna una instancia de una de estas subclases en lugar de una HttpResponse normal con el fin de significar un error, por ejemplo:

```
def mi_vista(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Pagina no encontrada</h1>')
    else:
        return HttpResponse('<h1>Pagina no encontrada</h1>')
```

Debido a que el error 404 es por mucho el error HTTP más común, hay una manera más fácil de manejarlo.

Cuando retornas un error tal como HttpResponseNotFound, eres responsable de definir el HTML de la página de error resultante:

```
return HttpResponseNotFound('<h1>Pagina no encontrada</h1>')
```

Por consistencia, y porque es una buena idea tener una página de error 404 consistente en todo tu sitio, Django provee una excepción Http404. Si tú emites una Http404 en cualquier punto de una vista de una función, Django la atrapará y retornará la página de error estándar de tu aplicación, junto con un código de error HTTP 404.

Éste es un ejemplo:

```
from django.http import Http404

def detalles(request, libro_id):
    try:
        p = Libro.objects.get(pk=libro_id)
    except Libro.DoesNotExist:
        raise Http404
    return render_to_response('biblioteca/detalles.html', {'poll': p})
```

Con el fin de usar la excepción Http404 al máximo, deberías crear una plantilla que se muestra cuando un error 404 es emitido. Esta plantilla debería ser llamada 404.html, y debería colocarse en el nivel superior de tu árbol de plantillas.

Personalizar la Vista 404 (Not Found)

Cuando tu emites una excepción Http404, Django carga una vista especial dedicada a manejar errores 404. Por omisión, es la vista django.views.defaults.page_not_found, la cual carga y renderiza la plantilla 404.html.

Esto significa que necesitas definir una plantilla 404.html en tu directorio raíz de plantillas. Esta plantilla será usada para todos los errores 404.

Esta vista page_not_found debería ser suficiente para el 99% de las aplicaciones Web, pero si tú quieres reemplazar la vista 404, puedes especificar handler404 en tu URLconf, de la siguiente manera:

```
from django.conf.urls import url

urlpatterns = [
    ...
]

handler404 = 'mysite.views.my_custom_404_view'
```

Detrás de escena, Django determina la vista 404 buscando por handler404. Por omisión, las URLconfs contienen la siguiente línea:

```
from django.conf.urls import url
```

Esto se encarga de establecer handler404 en el módulo actual. Como puedes ver en django/conf/urls/defaults.py, handler404 está fijado a 'django.views.defaults.page_not_found' por omisión.

Hay tres cosas para tener en cuenta sobre las vistas 404:

1. La vista 404 es llamada también si Django no encuentra una coincidencia después de verificar toda expresión regular en la URLconf.

2. Si no defines tu propia vista 404 – y simplemente usas la predeterminada, lo cual es recomendado – tú aún tienes una obligación: crear una plantilla 404.html en la raíz de tu directorio de plantillas. La vista 404 predeterminada usará esa plantilla para todos los errores 404.

3. Si DEBUG está establecido a True (en tu modulo de configuración), entonces tu vista 404 nunca será usada, y se mostrará en su lugar el trazado de pila.

Personalizar la Vista 500 (Server Error)

De manera similar, Django ejecuta un comportamiento especial en el caso de errores de ejecución en el código de la vista. Si una vista resulta en una excepción, Django llamará, de manera predeterminada, a la vista *django.views.defaults.server_error*, la cual carga y renderiza la plantilla 500.html.

Esto significa que necesitas definir una plantilla 500.html en el directorio raíz de plantillas. Esta plantilla será usada para todos los errores de servidor.

Esta vista *server_error* debería ser suficiente para el 99% de las aplicaciones Web, pero si tú quieres reemplazar la vista, puedes especificar handler500 en tu URLconf, de la siguiente manera:

```
from django.conf.urls import url

urlpatterns[
    ...
]

handler500 = 'mysite.views.my_custom_error_view'
```

Licencia y Copyrigth

Licencia de documentación libre de GNU

Versión 1.2, November 2002

Esta es una traducción no oficial de la GNU Free Document License (GFDL), versión 1.2 a Español (Castellano). No ha sido publicada por la Free Software Foundation y no establece legalmente los términos de distribución para trabajos que usen la GFDL (sólo el texto de la versión original en Inglés de la GFDL lo hace). Sin embargo, esperamos que esta traducción ayude los hispanohablantes a entender mejor la GFDL. La versión original de la GFDL está disponible en la Free Software Foundation (http://www.gnu.org/copyleft/fdl.html).

Esta traducción está basada en una de la versión 1.1 de Igor Támara y Pablo Reyes. Sin embargo la responsabilidad de su interpretación es de Joaquín Seoane.

Preámbulo

El propósito de esta Licencia es permitir que un manual, libro de texto, u otro documento escrito sea *libre* en el sentido de libertad: asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera comercial o no. En segundo término, esta Licencia proporciona al autor y al editor una manera de obtener reconocimiento por su trabajo, sin que se le considere responsable de las modificaciones realizadas por otros.

Esta Licencia es de tipo *copyleft*, lo que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Complementa la Licencia Pública General de GNU, que es una licencia tipo *copyleft* diseñada para el software libre.

Hemos diseñado esta Licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: un programa libre debe venir con manuales que ofrezcan las mismas libertades que el software. Pero esta licencia no se limita a manuales de software; puede usarse para cualquier texto, sin tener en cuenta su temática o si se publica como libro impreso o no. Recomendamos esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

1.0 APLICABILIDAD Y DEFINICIONES

Esta licencia se aplica a cualquier manual u otro trabajo, en cualquier soporte, que contenga una nota del propietario de los derechos de autor que indique que puede ser distribuido bajo los términos de esta licencia. Tal nota garantiza en cualquier lugar del mundo, sin pago de derechos y sin límite de tiempo, el uso de dicho trabajo según las condiciones aquí estipuladas.

En adelante la palabra Documento se referirá a cualquiera de dichos manuales o trabajos. Cualquier persona es un licenciatario y será referido como Usted. Usted acepta la licencia si copia, modifica o distribuye el trabajo de cualquier modo que requiera permiso según la ley de propiedad intelectual. Una Versión Modificada del Documento significa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones
y/o traducciones a otro idioma.

Una Sección Secundaria es un apéndice con título o una sección preliminar del Documento que trata exclusivamente de la relación entre los autores o editores y el tema general del Documento (o temas relacionados) pero que no contiene nada que entre directamente en dicho tema general (por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar nada de matemáticas). La relación puede ser una conexión histórica con el tema o temas relacionados, o una opinión legal, comercial, filosófica, ética o política acerca de ellos.

Las Secciones Invariantes son ciertas Secciones Secundarias cuyos títulos son designados como Secciones Invariantes en la nota que indica que el documento es liberado bajo esta Licencia. Si una sección no entra en la definición de Secundaria, no puede designarse como Invariante. El documento puede no tener Secciones Invariantes. Si el Documento no identifica las Secciones Invariantes, es que no las tiene.

Los Textos de Cubierta son ciertos pasajes cortos de texto que se listan como Textos de Cubierta Delantera o Textos de Cubierta Trasera en la nota que indica que el documento es liberado bajo esta Licencia. Un Texto de Cubierta Delantera puede tener como mucho 5 palabras, y uno de Cubierta Trasera puede tener hasta 25 palabras. Una copia Transparente del Documento, significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público en general, apto para que los contenidos puedan ser vistos y editados directamente con editores de texto genéricos o (para imágenes compuestas por puntos) con programas genéricos de manipulación de imágenes o (para dibujos) con algún editor de dibujos ampliamente disponible, y que sea adecuado como entrada para formateadores de texto o para su traducción automática a formatos adecuados para formateadores de texto.

Una copia hecha en un formato definido como Transparente, pero cuyo marcaje o ausencia de él haya sido diseñado para impedir o dificultar modificaciones posteriores por parte de los lectores no es Transparente. Un formato de imagen no es Transparente si se usa para una cantidad de texto sustancial. Una copia que no es Transparente se denomina Opaca.

Como ejemplos de formatos adecuados para copias Transparentes están ASCII puro sin marcaje, formato de entrada de Texinfo, formato de entrada de LaTeX, SGML o XML usando una DTD disponible públicamente, y HTML, PostScript o PDF simples, que sigan los estándares y diseñados para que los modifiquen personas.

Ejemplos de formatos de imagen transparentes son PNG, XCF y JPG. Los formatos Opacos incluyen formatos propietarios que pueden ser leídos y editados únicamente en procesadores de palabras propietarios, SGML o XML para los cuáles las DTD y/o

herramientas de procesamiento no estén ampliamente disponibles, y HTML, PostScript o PDF generados por algunos procesadores de palabras sólo como salida.

La Portada significa, en un libro impreso, la página de título, más las páginas siguientes que sean necesarias para mantener legiblemente el material que esta Licencia requiere en la portada. Para trabajos en formatos que no tienen página de portada como tal, Portada significa el texto cercano a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del texto. Una sección Titulada XYZ significa una parte del Documento cuyo título es precisamente XYZ o contiene XYZ entre paréntesis, a continuación de texto que traduce XYZ a otro idioma (aquí XYZ se refiere a nombres de sección específicos mencionados más abajo, como Agradecimientos, Dedicatorias, Aprobaciones o Historia. Conservar el Título de tal sección cuando se modifica el Documento significa que permanece una sección titulada XYZ según esta definición.

El Documento puede incluir Limitaciones de Garantía cercanas a la nota donde se declara que al Documento se le aplica esta Licencia. Se considera que estas Limitaciones de Garantía están incluidas, por referencia, en la Licencia, pero sólo en cuanto a limitaciones de garantía: cualquier otra implicación que estas Limitaciones de Garantía puedan tener es nula y no tiene efecto en el significado de esta Licencia.

2. Copia literal

Usted puede copiar y distribuir el Documento en cualquier soporte, sea en forma comercial o no, siempre y cuando esta Licencia, las notas de copyright y la nota que indica que esta Licencia se aplica al Documento se reproduzcan en todas las copias y que usted no añada ninguna otra condición a las expuestas en esta Licencia. Usted no puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

Usted también puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias públicamente.

3. Copiado en cantidad

Si publica copias impresas del Documento (o copias en soportes que tengan normalmente cubiertas impresas) que sobrepasen las 100, y la nota de licencia del Documento exige Textos de Cubierta, debe incluir las copias con cubiertas que lleven en forma clara y legible todos esos Textos de Cubierta: Textos de Cubierta Delantera en la cubierta delantera y Textos de Cubierta Trasera en la cubierta trasera. Ambas cubiertas deben identificarlo a Usted clara y legiblemente como editor de tales copias.

La cubierta debe mostrar el título completo con todas las palabras igualmente prominentes y visibles. Además puede añadir otro material en las cubiertas. Las copias con cambios limitados a las cubiertas, siempre que conserven el título del Documento y satisfagan estas condiciones, pueden considerarse como copias literales.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros (tantos como sea razonable colocar) en la verdadera cubierta y situar el resto en páginas adyacentes. Si Usted pública o distribuye copias Opacas del Documento cuya cantidad exceda las 100, debe incluir una copia Transparente, que pueda ser le da por una máquina, con cada copia

Opaca, o bien mostrar, en cada copia Opaca, una dirección de red donde cualquier usuario de la misma tenga acceso por medio de protocolos públicos y estandarizados a una copia Transparente del Documento completa, sin material adicional. Si usted hace uso de la última opción, deberá tomar las medidas necesarias, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio establecido por lo menos un año después de la última vez que distribuya una copia Opaca de esa edición al público (directamente o a través de sus agentes o distribuidores).

Se solicita, aunque no es requisito, que se ponga en contacto con los autores del Documento antes de redistribuir gran número de copias, para darles la oportunidad de que le proporcionen una versión actualizada del Documento.

4. Modificaciones

Puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las secciones 2 y 3 anteriores, siempre que usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada haciendo el rol del Documento, por lo tanto dando licencia de distribución y modificación de la Versión Modificada a quienquiera posea una copia de la misma.

Además, debe hacer lo siguiente en la Versión Modificada:

- Usar en la Portada (y en las cubiertas, si hay alguna) un título distinto al del Documento y de sus versiones anteriores (que deberían, si hay alguna, estar listadas en la sección de Historia del Documento). Puede usar el mismo título de versiones anteriores al original siempre y cuando quien las publicó originalmente otorgue permiso.

- Listar en la Portada, como autores, una o más personas o entidades responsables de la autoría de las modificaciones de la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (todos sus autores principales, si hay menos de cinco), a menos que le eximan de tal requisito.

- Mostrar en la Portada como editor el nombre del editor de la Versión Modificada.

- Conservar todas las notas de copyright del Documento.

- Añadir una nota de copyright apropiada a sus modificaciones, adyacente a las otras notas de copyright.

- Incluir, inmediatamente después de las notas de copyright, una nota de licencia dando el permiso para usar la Versión Modificada bajo los términos de esta Licencia, como se muestra en la Adenda al final de este documento.

- Conservar en esa nota de licencia el listado completo de las Secciones Invariantes y de los Textos de Cubierta que sean requeridos en la nota de Licencia del Documento original.

- Incluir una copia sin modificación de esta Licencia.

- Conservar la sección Titulada Historia, conservar su Título y añadirle un elemento que declare al menos el título, el año, los nuevos autores y el editor de la Versión Modificada, tal como figuran en la Portada. Si no hay una sección Titulada Historia en el Documento, crear una estableciendo el título, el año, los autores y el editor del Documento, tal como figuran en su Portada, añadiendo además un elemento describiendo la Versión Modificada, comose estableció en la oración anterior.

- Conservar la dirección en red, si la hay, dada en el Documento para el acceso público a una copia Transparente del mismo, así como las otras direcciones de red dadas en el Documento para versiones anteriores en las que estuviese basado. Pueden ubicarse en la sección Historia. Se puede omitir la ubicación en red de un trabajo que haya sido publicado por lo menos cuatro años antes que el Documento mismo, o si el editor original de dicha versión da permiso.

- En cualquier sección Titulada Agradecimientos o Dedicatorias, Conservar el Título de la sección y conservar en ella toda la sustancia y el tono de los agradecimientos y/o dedicatorias incluidas por cada contribuyente.

- Conservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Números de sección o el equivalente no son considerados parte de los títulos de la sección.
- Borrar cualquier sección titulada Aprobaciones. Tales secciones no pueden estar incluidas en las Versiones Modificadas.

- No cambiar el título de ninguna sección existente a Aprobaciones ni a uno que entre en conflicto con el de alguna Sección Invariante.

- Conservar todas las Limitaciones de Garantía.

Si la Versión Modificada incluye secciones o apéndices nuevos que califiquen como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, añada sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.

Puede añadir una sección titulada Aprobaciones, siempre que contenga únicamente aprobaciones de su Versión Modificada por otras fuentes –por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como la definición oficial de un estándar.

Puede añadir un pasaje de hasta cinco palabras como Texto de Cubierta Delantera y un pasaje de hasta 25 palabras como Texto de Cubierta Trasera en la Versión Modificada. Una entidad solo puede añadir (o hacer que se añada) un pasaje al Texto de Cubierta Delantera y uno al de Cubierta Trasera. Si el Documento ya incluye unos textos de cubiertas añadidos previamente por usted o por la misma entidad que usted representa, usted no puede añadir otro; pero puede reemplazar el anterior, con permiso explícito del editor que agregó el texto anterior.

Con esta Licencia ni los autores ni los editores del Documento dan permiso para usar sus nombres para publicidad ni para asegurar o implicar aprobación de cualquier Versión Modificada.

5. Combinación de documentos

Usted puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, listadas todas como Secciones Invariantes del trabajo combinado en su nota de licencia. Asimismo debe incluir la Limitación de Garantía.

El trabajo combinado necesita contener solamente una copia de esta Licencia, y puede reemplazar varias Secciones Invariantes idénticas por una sola copia. Si hay varias Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único añadiéndole al final del mismo, entre paréntesis, el nombre del autor o editor original de esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes de la nota de licencia del trabajo combinado.

En la combinación, debe combinar cualquier sección Titulada Historia de los documentos originales, formando una sección Titulada Historia; de la misma forma combine cualquier sección Titulada Agradecimientos, y cualquier sección Titulada Dedicatorias. Debe borrar todas las secciones tituladas Aprobaciones.

6. Colecciones de documentos

Puede hacer una colección que conste del Documento y de otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en todos los documentos por una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para cada copia literal de cada uno de los documentos en cualquiera de los demás aspectos.

Puede extraer un solo documento de una de tales colecciones y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraído, y siga esta Licencia en todos los demás aspectos relativos a la copia literal de dicho documento.

7. Agregación con trabajos independientes

Una recopilación que conste del Documento o sus derivados y de otros documentos o trabajos separados e independientes, en cualquier soporte de almacenamiento o distribución, se denomina un agregado si el copyright resultante de la compilación no se usa para limitar los derechos de los usuarios de la misma más allá de lo que los de los trabajos individuales permiten.

Cuando el Documento se incluye en un agregado, esta Licencia no se aplica a otros trabajos del agregado que no sean en sí mismos derivados del Documento.

Si el requisito de la sección 3 sobre el Texto de Cubierta es aplicable a estas copias del Documento y el Documento es menor que la mitad del agregado entero, los Textos de Cubierta del Documento pueden colocarse en cubiertas que enmarquen solamente el Documento dentro del agregado, o el equivalente electrónico de las cubiertas si el documento está en forma electrónica.

En caso contrario deben aparecer en cubiertas impresas enmarcando todo el agregado.

8. Traducción

La Traducción es considerada como un tipo de modificación, por lo que usted puede distribuir traducciones del Documento bajo los términos de la sección 4. El reemplazo de las Secciones Invariantes con traducciones requiere permiso especial de los dueños de derecho de autor, pero usted puede añadir traducciones de algunas o todas las Secciones Invariantes a las versiones originales de las mismas.

Puede incluir una traducción de esta Licencia, de todas las notas de licencia del documento, así como de las Limitaciones de Garantía, siempre que incluya también la versión en Inglés de esta Licencia y las versiones originales de las notas de licencia y Limitaciones de Garantía. En caso de desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la nota de licencia o la limitación de garantía, la versión original en Inglés prevalecerá.

Si una sección del Documento está Titulada Agradecimientos, Dedicatorias o Historia el requisito (sección 4) de Conservar su Título (Sección 1) requerirá, típicamente, cambiar su título.

9. Terminación

Usted no puede copiar, modificar, sublicenciar o distribuir el Documento salvo por lo permitido expresamente por esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y dará por terminados automáticamente sus derechos bajo esa Licencia. Sin embargo, los terceros que hayan recibido copias, o derechos, de usted bajo esta Licencia no verán terminadas sus licencias, siempre que permanezcan en total conformidad con ella.

10. Revisiones futuras de esta licencia

De vez en cuando la Free Software Foundation puede publicar versiones nuevas y revisadas de la Licencia de Documentación Libre GNU. Tales versiones nuevas serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar nuevos problemas o intereses. Vea http://www.gnu.org/copyleft/.

Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que se aplica una versión numerada en particular de esta licencia o cualquier versión posterior, usted tiene la opción de seguir los términos y condiciones de la versión especificada o cualquiera posterior que haya sido publicada (no como borrador) por la Free Software Foundation.

Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada (no como borrador) por la Free Software Foundation.

11.- ADENDA: Cómo usar esta Licencia en sus documentos

Para usar esta licencia en un documento que usted haya escrito, incluya una copia de la Licencia en el documento y ponga el siguiente copyright y nota de licencia justo después de la página de título:

Si tiene Secciones Invariantes, Textos de Cubierta Delantera y Textos de Cubierta Trasera, reemplace la frase sin... Trasera por esto:

siendo las Secciones Invariantes LISTE SUS TÍTULOS, siendo los Textos de Cubierta Delantera LISTAR, y siendo sus Textos de Cubierta Trasera LISTAR.

Si tiene Secciones Invariantes sin Textos de Cubierta o cualquier otra combinación de los tres, mezcle ambas alternativas para adaptarse a la situación.

Si su documento contiene ejemplos de código de programa no triviales, recomendamos liberar estos ejemplos en paralelo bajo la licencia de software libre que usted elija, como la Licencia Pública General de GNU (GNU General Public License), para permitir su uso en software libre.