El libro de Django
Autores: Adrian Holovaty y Jacob Kaplan-Moss
Editor técnico: Jeremy Dunck

NOTA: Este trabajo de traducción aún no está completo.
Número de revisión: 757
Fecha de compaginación: 28 de julio de 2008

Índice general

Preliminares XVII
Reconocimientos . . . XVII
Sobre los autores . . . XVII
Sobre el editor técnico . . . XVII
Sobre los traductores . . . XVIII
Sobre el libro . . . XVIII

Introducción XIX

1. Introducción a Django 1
1.1. ¿Qué es un Framework Web? . . . 1
1.2. El patrón de diseño MVC . . . 2
1.3. La historia de Django . . . 3
1.4. Cómo leer este libro . . . 4
1.4.1. Conocimientos de programación requeridos . . . 5
1.4.2. Conocimientos de Python requeridos . . . 5
1.4.3. Nuevas características de Django . . . 5
1.4.4. Obteniendo ayuda . . . 5
1.5. ¿Qué sigue? . . . 5

2. Empezando 7
2.1. Instalar Python . . . 7
2.2. Instalar Django . . . 7
2.2.1. Instalar un lanzamiento oficial . . . 7
2.2.2. Instalar Django desde Subversion . . . 8
2.3. Configurando la base de datos . . . 8
2.3.1. Usar Django con PostgreSQL . . . 9
2.3.2. Usar Django con SQLite 3 . . . 9
2.3.3. Usar Django con MySQL . . . 9
2.3.4. Usar Django sin una base de datos . . . 9
2.4. Comenzando un proyecto . . . 9
2.4.1. El servidor de desarrollo . . . 10
2.5. ¿Qué sigue? . . . 11

3. Los principios de las páginas Web dinámicas 13
3.1. Tu primera Vista: Contenido dinámico . . . 13
3.2. Mapeando URLs a Vistas . . . 14
3.3. Cómo procesa una petición Django . . . 16
3.3.1. Cómo procesa una petición Django: Detalles completos . . . 16
3.4. URLconfs y el acoplamiento débil . . . 18
3.5. Errores 404 . . . 18
3.6. Tu segunda Vista: URLs dinámicas . . . 20
3.6.1. Algunas palabras acerca de las URLs bonitas . . . 20
3.6.2. Comodines en los patrones URL . . . 20
3.7. Páginas de error bonitas con Django . . . 22
3.8. ¿Qué sigue? . . . 23

4. El sistema de plantillas de Django **25**
4.1. Sistema básico de plantillas . . . 25
4.2. Empleo del sistema de plantillas . . . 26
4.2.1. Creación de objetos Template . . . 26
4.2.2. Renderizar una plantilla . . . 28
4.2.3. Múltiples contextos, mismas plantillas . . . 29
4.2.4. Búsqueda del contexto de una variable . . . 29
4.2.5. Jugando con objetos Context . . . 32
4.3. Etiquetas de plantillas básicas y filtros . . . 32
4.3.1. Etiquetas . . . 32
4.3.2. Filtros . . . 36
4.4. Filosofía y Limitaciones . . . 37
4.5. Uso de plantillas en las vistas . . . 38
4.6. Cargadores de plantillas . . . 39
4.6.1. render_to_response() . . . 42
4.6.2. El truco locals() . . . 42
4.6.3. Subdirectorios en get_template() . . . 43
4.6.4. La etiqueta de plantilla `include` . . . 43
4.7. Herencia de plantillas . . . 43
4.8. ¿Qué sigue? . . . 46

5. Interactuar con una base de datos: Modelos **47**
5.1. La manera "tonta" de hacer una consulta a la base de datos en las vistas . . . 47
5.2. El patrón de diseño MTV . . . 48
5.3. Configuración de la base de datos . . . 49
5.4. Tu primera aplicación . . . 51
5.5. Definir modelos en Python . . . 52
5.6. Tu primer modelo . . . 52
5.7. Instalando el modelo . . . 54
5.8. Acceso básico a datos . . . 56
5.9. Agregando strings de representación del modelo . . . 56
5.10. Insertando y actualizando datos . . . 57
5.11. Seleccionar objetos . . . 58
5.11.1. Filtrar datos . . . 59
5.11.2. Obteniendo objetos individuales . . . 60
5.11.3. Ordenando datos . . . 60
5.11.4. Encadenando búsquedas . . . 61
5.11.5. Rebanando datos . . . 61
5.12. Eliminando objetos . . . 62
5.13. Realizando cambios en el esquema de una base de datos . . . 62
5.13.1. Agregando campos . . . 62
5.13.2. Eliminando campos . . . 64
5.13.3. Eliminando campos Many-to-Many . . . 64
5.13.4. Eliminando modelos . . . 64
5.14. ¿Qué sigue? . . . 65

6. El sitio de Administración Django **67**
6.1. Activando la interfaz de administración . . . 67
6.2. Usando la interfaz de administración . . . 68
6.2.1. Usuarios, Grupos y Permisos . . . 73
6.3. Personalizando la interfaz de administración . . . 73
6.4. Personalizando la apariencia de la interfaz de administración . . . 77
6.5. Personalizando la página índice del administrador . . . 79
6.6. Cuándo y porqué usar la interfaz de administración . . . 79
6.7. ¿Qué sigue? . . . 79

7. Procesamiento de formularios **81**
7.1. Búsquedas . . . 81
7.2. El "formulario perfecto" . . . 83
7.3. Creación de un formulario para comentarios . . . 83
7.4. Procesamiento de los datos suministrados . . . 86
7.5. Nuestras propias reglas de validación . . . 87
7.6. Una presentación personalizada . . . 88
7.7. Creando formularios a partir de Modelos . . . 89
7.8. ¿Qué sigue? . . . 90

8. Vistas avanzadas y URLconfs **91**
8.1. Trucos de URLconf . . . 91
8.1.1. Importación de funciones de forma efectiva . . . 91
8.1.2. Usar múltiples prefijos de vista . . . 92
8.1.3. Casos especiales de URLs en modo Debug . . . 93
8.1.4. Usar grupos con nombre . . . 93
8.1.5. Comprender el algoritmo de combinación/agrupación . . . 95
8.1.6. Pasarle opciones extra a las funciones vista . . . 95
8.1.7. Usando argumentos de vista por omisión . . . 99
8.1.8. Manejando vistas en forma especial . . . 100
8.1.9. Capturando texto en URLs . . . 100
8.1.10. Entendiendo dónde busca una URLconf . . . 101
8.2. Incluyendo otras URLconfs . . . 101
8.2.1. Cómo trabajan los parámetros capturados con include() . . . 102
8.2.2. Cómo funcionan las opciones extra de URLconf con include() . . . 102
8.3. ¿Qué sigue? . . . 103

9. Vistas genéricas **105**
9.1. Usar vistas genéricas . . . 105
9.2. Vistas genéricas de objetos . . . 106
9.3. Extender las vistas genéricas . . . 107
9.3.1. Crear contextos de plantilla "amistosos" . . . 108
9.3.2. Agregar un contexto extra . . . 108
9.3.3. Mostrar subconjuntos de objetos . . . 109
9.3.4. Filtrado complejo con funciones wrapper . . . 109
9.3.5. Realizar trabajo extra . . . 110
9.4. ¿Qué sigue? . . . 111

10. Extendiendo el sistema de plantillas **113**
10.1. Revisión del lenguaje de plantillas . . . 113
10.2. Procesadores de contexto . . . 113
10.2.1. django.core.context_processors.auth . . . 117
10.2.2. django.core.context_processors.debug . . . 117
10.2.3. django.core.context_processors.i18n . . . 117
10.2.4. django.core.context_processors.request . . . 117
10.2.5. Consideraciones para escribir tus propios procesadores de contexto . . . 117
10.3. Detalles internos de la carga de plantillas . . . 118
10.4. Extendiendo el sistema de plantillas . . . 119
10.4.1. Creando una biblioteca para plantillas . . . 119
10.4.2. Escribiendo filtros de plantilla personalizados . . . 120
10.4.3. Escribiendo etiquetas de plantilla personalizadas . . . 121
10.4.4. Un atajo para etiquetas simples . . . 125
10.4.5. Etiquetas de inclusión . . . 125
10.5. Escribiendo cargadores de plantillas personalizados . . . 126
10.6. Usando la referencia de plantillas incorporadas . . . 127
10.7. Configurando el sistema de plantillas en modo autónomo . . . 128
10.8. ¿Qué sigue? . . . 128

11. Generación de contenido no HTML **129**
11.1. Lo básico: Vistas y tipos MIME . . . 129
11.2. Producción de CSV . . . 130
11.3. Generando PDFs . . . 131
11.3.1. Instalando ReportLab . . . 131
11.3.2. Escribiendo tu Vista . . . 131
11.3.3. PDFs complejos . . . 132
11.4. Otras posibilidades . . . 133
11.5. El Framework de Feeds de Sindicación . . . 133
11.5.1. Inicialización . . . 133
11.5.2. Un Feed simple . . . 134
11.5.3. Un Feed más complejo . . . 135
11.5.4. Especificando el tipo de Feed . . . 136
11.5.5. Enclosures . . . 137
11.5.6. Idioma . . . 137
11.5.7. URLs . . . 137
11.5.8. Publicando feeds Atom y RSS conjuntamente . . . 137
11.6. El framework Sitemap . . . 138
11.6.1. Instalación . . . 138
11.6.2. Inicialización . . . 139
11.6.3. Clases Sitemap . . . 139
11.6.4. Accesos directos . . . 140
11.6.5. Creando un índice Sitemap . . . 141
11.6.6. Haciendo ping a Google . . . 141
11.7. ¿Qué sigue? . . . 142

12. Sesiones, usuario e inscripciones **143**
12.1. Cookies . . . 143
12.1.1. Cómo definir y leer los valores de las cookies . . . 144
12.1.2. Las cookies tienen doble filo . . . 145
12.2. El entorno de sesiones de Django . . . 145
12.2.1. Activar sesiones . . . 146
12.2.2. Usar las sesiones en una vista . . . 146
12.2.3. Comprobar que las *cookies* sean utilizables . . . 147
12.2.4. Usar las sesiones fuera de las vistas . . . 148
12.2.5. Cuándo se salvan las sesiones . . . 148
12.2.6. Sesiones breves frente a sesiones persistentes . . . 149
12.2.7. Otras características de las sesiones . . . 149
12.3. Usuarios e identificación . . . 150
12.3.1. Habilitando el soporte para autentificación . . . 151
12.4. Utilizando usuarios . . . 151
12.4.1. Iniciar y cerrar sesión . . . 153
12.4.2. Limitar el acceso a los usuarios identificados . . . 154
12.4.3. Limitar el acceso a usuarios que pasan una prueba . . . 155
12.4.4. Gestionar usuarios, permisos y grupos . . . 156
12.4.5. Usar información de autentificación en plantillas . . . 158
12.5. El resto de detalles: permisos, grupos, mensajes y perfiles . . . 159
12.5.1. Permisos . . . 159
12.5.2. Grupos . . . 160
12.5.3. Mensajes . . . 160
12.5.4. Perfiles . . . 161
12.6. ¿Qué sigue? . . . 161

13. Cache **163**
13.1. Activando el Cache . . . 163
13.1.1. Memcached . . . 164
13.1.2. Cache en Base de datos . . . 164
13.1.3. Cache en Sistema de Archivos . . . 165
13.1.4. Cache en Memoria local . . . 165
13.1.5. Cache Simple (para desarrollo) . . . 165
13.1.6. Cache Dummy (o estúpida) . . . 165
13.1.7. Argumentos de CACHE_BACKEND . . . 165
13.2. La cache por sitio . . . 166
13.3. Cache por vista . . . 167
13.3.1. Especificando la cache por vista en URLconf . . . 167
13.4. La API de cache de bajo nivel . . . 168
13.5. Caches upstream . . . 169
13.5.1. Usando el encabezado Vary . . . 169
13.5.2. Otros Encabezados de cache . . . 170
13.6. Otras optimizaciones . . . 171
13.7. Orden de MIDDLEWARE_CLASSES . . . 171
13.8. ¿Qué sigue? . . . 172

14. Otros sub-frameworks contribuidos **173**
14.1. La biblioteca estándar de Django . . . 173
14.2. Sites . . . 174
14.2.1. Escenario 1: reuso de los datos en múltiples sitios . . . 174
14.2.2. Escenario 2: alojamiento del nombre/dominio de tu sitio en un solo lugar . . . 174
14.2.3. Modo de uso del *framework sites* . . . 174
14.2.4. Las capacidades del framework Sites . . . 175
14.2.5. CurrentSiteManager . . . 177
14.2.6. El uso que hace Django del *framework* Sites . . . 178
14.3. Flatpages . . . 178
14.3.1. Usando flatpages . . . 179
14.3.2. Agregando, modificando y eliminando flatpages . . . 180
14.3.3. Usando plantillas de flatpages . . . 180
14.4. Redirects . . . 181
14.4.1. Usando el framework redirects . . . 181
14.4.2. Agregando, modificando y eliminando redirecciones . . . 181
14.5. Protección contra CSRF . . . 182
14.5.1. Un ejemplo simple de CSRF . . . 182
14.5.2. Un ejemplo más complejo de CSRF . . . 182
14.5.3. Previniendo la CSRF . . . 183
14.6. Haciendo los datos más humanos . . . 184
14.6.1. apnumber . . . 184
14.6.2. intcomma . . . 184
14.6.3. intword . . . 184
14.6.4. ordinal . . . 184
14.7. Filtros de marcado . . . 185
14.8. ¿Qué sigue? . . . 185

15. Middleware **187**
15.1. Qué es middleware . . . 187
15.2. Instalación de Middleware . . . 188
15.3. Métodos de un Middleware . . . 188
15.3.1. Inicializar: __init__(self) . . . 188
15.3.2. Pre-procesador de petición: process_request(self, request) . . . 188
15.3.3. Pre-procesador de vista: process_view(self, request, view, args, kwargs) . . . 189
15.3.4. Pos-procesador de respuesta: process_response(self, request, response) . . . 189
15.3.5. Pos-procesador de excepción: process_exception(self, request, exception) . . . 189
15.4. Middleware incluido . . . 190
15.4.1. Middleware de soporte para autenticación . . . 190

15.4.2. Middleware "Common" . . . 190
15.4.3. Middleware de compresión . . . 190
15.4.4. Middleware de GET condicional . . . 191
15.4.5. Soporte para uso de proxy inverso (Middleware X-Forwarded-For) . . . 191
15.4.6. Middleware de soporte para sesiones . . . 191
15.4.7. Middleware de cache de todo el sitio . . . 191
15.4.8. Middleware de transacción . . . 191
15.4.9. Middleware "X-View" . . . 191
15.5. ¿Qué sigue? . . . 191

16. Integración con Base de datos y Aplicaciones existentes 193
16.1. Integración con una base de datos existente . . . 193
16.1.1. Empleo de `inspectdb` . . . 193
16.1.2. Limpiar los modelos generados . . . 194
16.2. Integración con un sistema de autentificación . . . 194
16.2.1. Especificar los back-ends de autentificación . . . 195
16.2.2. Escribir un back-end de autentificación . . . 195
16.3. Integración con aplicaciones web existentes . . . 196
16.4. ¿Qué sigue? . . . 196

17. Extendiendo la Interfaz de Administración de Django 199
17.1. El Zen de la aplicación Admin . . . 200
17.1.1. "Usuarios confiables ..." . . . 200
17.1.2. "... editando ..." . . . 200
17.1.3. "... contenido estructurado" . . . 200
17.1.4. Parada Completa . . . 200
17.2. Pesonalizando las plantillas de la interfaz . . . 201
17.2.1. Plantillas de modelos propios . . . 201
17.2.2. JavaScript Personalizado . . . 203
17.3. Creando vistas de administración personalizadas . . . 203
17.4. Sobreescribiendo vistas incorporadas . . . 205
17.5. ¿Qué sigue? . . . 205

18. Internacionalización 207
18.1. Especificando cadenas de traducción en código Python . . . 208
18.1.1. Funciones estándar de traducción . . . 208
18.1.2. Marcando cadenas como no-op . . . 209
18.1.3. Traducción perezosa . . . 209
18.1.4. Pluralización . . . 209
18.2. Especificando cadenas de traducción en código de plantillas . . . 210
18.3. Creando archivos de idioma . . . 211
18.3.1. Creando los archivos de mensajes . . . 211
18.3.2. Compilando archivos de mensajes . . . 212
18.4. Cómo descubre Django la preferencia de idioma . . . 212
18.5. La vista de redirección set_language . . . 214
18.6. Usando traducciones en tus propios proyectos . . . 214
18.7. Traducciones y JavaScript . . . 215
18.7.1. La vista javascript_catalog . . . 215
18.7.2. Usando el catálogo de traducciones JavaScript . . . 216
18.7.3. Creando catálogos de traducciones JavaScript . . . 216
18.8. Notas para usuarios familiarizados con `gettext` . . . 217
18.9. ¿Qué sigue? . . . 217

19. Seguridad 219
19.1. El tema de la seguridad en la Web . . . 219
19.2. Inyección de SQL . . . 219
19.2.1. La solución . . . 220
19.3. Cross-Site Scripting (XSS) . . . 221
19.3.1. La solución . . . 221
19.4. Cross-Site Request Forgery . . . 222
19.5. Session Forging/Hijacking . . . 222
19.5.1. La solución . . . 223
19.6. Inyección de cabeceras de email . . . 223
19.6.1. La solución . . . 224
19.7. Directory Traversal . . . 224
19.7.1. La solución . . . 224
19.8. Exposición de mensajes de error . . . 225
19.8.1. La solución . . . 225
19.9. Palabras finales sobre la seguridad . . . 225
19.10¿Qué sigue? . . . 226

20. Implementando Django 227
20.1. Nada Compartido . . . 227
20.2. Un nota sobre preferencias personales . . . 228
20.3. Usando Django con Apache y mod_python . . . 229
20.3.1. Configuración básica . . . 229
20.3.2. Corriendo multiples instalaciones de Django en la misma instancia Apache . . . 230
20.3.3. Corriendo un servidor de desarrollo con mod_python . . . 231
20.3.4. Sirviendo Django y archivos multimedia desde la misma instancia Apache . . . 231
20.3.5. Manejo de errores . . . 232
20.3.6. Manejando fallas de segmentación . . . 232
20.4. Usando Django con FastCGI . . . 232
20.4.1. Descripción de FastCGI . . . 232
20.4.2. Ejecutando tu Servidor FastCGI . . . 233
20.4.3. Usando Django con Apache y FastCGI . . . 234
20.4.4. FastCGI y lighttpd . . . 235
20.4.5. Ejecutando Django en un Proveedor de Hosting Compartido con Apache . . . 236
20.5. Escalamiento . . . 236
20.5.1. Ejecutando en un Servidor Único . . . 237
20.5.2. Separando el Servidor de Bases de Datos . . . 237
20.5.3. Ejecutando un Servidor de Medios Separado . . . 237
20.5.4. Implementando Balance de Carga y Redundancia . . . 239
20.5.5. Yendo a lo grande . . . 239
20.6. Ajuste de Performance . . . 242
20.6.1. No hay tal cosa como demasiada RAM . . . 242
20.6.2. Deshabilita Keep-Alive . . . 242
20.6.3. Usa memcached . . . 242
20.6.4. Usa memcached siempre . . . 242
20.6.5. Únete a la Conversación . . . 243
20.7. ¿Qué sigue? . . . 243

A. Casos de estudio 245
A.1. Elenco . . . 245
A.2. ¿Por qué Django? . . . 246
A.3. Comenzando . . . 247
A.4. Portando código existente . . . 247
A.5. ¿Cómo les fue? . . . 247
A.6. Estructura de Equipo . . . 249
A.7. Implementación . . . 249

B. Referencia de la definición de modelos **251**
B.1. Campos . . . 251
B.1.1. AutoField . . . 252
B.1.2. BooleanField . . . 252
B.1.3. CharField . . . 252
B.1.4. CommaSeparatedIntegerField . . . 252
B.1.5. DateField . . . 252
B.1.6. DateTimeField . . . 252
B.1.7. EmailField . . . 252
B.1.8. FileField . . . 252
B.1.9. FilePathField . . . 253
B.1.10. FloatField . . . 254
B.1.11. ImageField . . . 254
B.1.12. IntegerField . . . 254
B.1.13. IPAddressField . . . 254
B.1.14. NullBooleanField . . . 254
B.1.15. PhoneNumberField . . . 254
B.1.16. PositiveIntegerField . . . 254
B.1.17. PositiveSmallIntegerField . . . 254
B.1.18. SlugField . . . 255
B.1.19. SmallIntegerField . . . 255
B.1.20. TextField . . . 255
B.1.21. TimeField . . . 255
B.1.22. URLField . . . 255
B.1.23. USStateField . . . 255
B.1.24. XMLField . . . 255
B.2. Opciones Universales de Campo . . . 255
B.2.1. null . . . 256
B.2.2. blank . . . 256
B.2.3. choices . . . 256
B.2.4. db_column . . . 257
B.2.5. db_index . . . 257
B.2.6. default . . . 257
B.2.7. editable . . . 257
B.2.8. help_text . . . 257
B.2.9. primary_key . . . 257
B.2.10. radio_admin . . . 257
B.2.11. unique . . . 257
B.2.12. unique_for_date . . . 257
B.2.13. unique_for_month . . . 258
B.2.14. unique_for_year . . . 258
B.2.15. verbose_name . . . 258
B.3. Relaciones . . . 258
B.3.1. Relaciones Muchos-a-Uno . . . 258
B.3.2. Relaciones Muchos-a-Muchos . . . 260
B.4. Opciones de los Metadatos del Modelo . . . 261
B.4.1. db_table . . . 261
B.4.2. get_latest_by . . . 262
B.4.3. order_with_respect_to . . . 262
B.4.4. ordering . . . 262
B.4.5. permissions . . . 262
B.4.6. unique_together . . . 263
B.4.7. verbose_name . . . 263
B.4.8. verbose_name_plural . . . 263
B.5. Managers . . . 263
B.5.1. Nombres de Manager . . . 264
B.5.2. Managers Personalizados . . . 264
B.6. Métodos de Modelo . . . 266

B.6.1. __str__ . . . 266
B.6.2. get_absolute_url . . . 267
B.6.3. Ejecutando SQL personalizado . . . 268
B.6.4. Sobreescribiendo los Métodos por omisión del Modelo . . . 268
B.7. Opciones del Administrador . . . 268
B.7.1. date_hierarchy . . . 269
B.7.2. fields . . . 269
B.7.3. js . . . 270
B.7.4. list_display . . . 270
B.7.5. list_display_links . . . 271
B.7.6. list_filter . . . 272
B.7.7. list_per_page . . . 272
B.7.8. list_select_related . . . 272
B.7.9. ordering . . . 272
B.7.10. save_as . . . 272
B.7.11. save_on_top . . . 272
B.7.12. search_fields . . . 272

C. Referencia de la API de base de datos **275**
C.1. Creando Objetos . . . 275
C.1.1. Qué pasa cuando grabas? . . . 276
C.1.2. Claves primarias autoincrementales . . . 276
C.2. Grabando cambios de objetos . . . 277
C.3. Recuperando objetos . . . 277
C.4. Caching y QuerySets . . . 278
C.5. Filtrando objetos . . . 278
C.5.1. Encadenando filtros . . . 279
C.5.2. Limitando QuerySets . . . 280
C.5.3. Métodos de consulta que retornan nuevos QuerySets . . . 280
C.5.4. Metodos de `QuerySet` que no devuelven un `QuerySet` . . . 283
C.6. Patrones de búsqueda . . . 285
C.6.1. exact . . . 286
C.6.2. iexact . . . 286
C.6.3. contains . . . 286
C.6.4. icontains . . . 286
C.6.5. gt, gte, lt, and lte . . . 287
C.6.6. in . . . 287
C.6.7. startswith . . . 287
C.6.8. istartswith . . . 287
C.6.9. endswith and iendswith . . . 287
C.6.10. range . . . 287
C.6.11. year, month, and day . . . 287
C.6.12. isnull . . . 288
C.6.13. search . . . 288
C.6.14. El patrón de búsqueda pk . . . 288
C.7. Búsquedas complejas con Objetos Q . . . 289
C.8. Objetos Relacionados . . . 289
C.8.1. Consultas Que Cruzan Relaciones . . . 290
C.8.2. Relaciones de Clave Foránea . . . 290
C.8.3. Relaciones de Clave Foreánea "Inversas" . . . 290
C.8.4. Relaciones muchos-a-muchos . . . 292
C.8.5. Consultas que Abarcan Objetos Relacionados . . . 292
C.9. Borrando Objectos . . . 292
C.10. Métodos de Instancia Adicionales . . . 293
C.10.1. get_FOO_display() . . . 293
C.10.2. get_next_by_FOO(**kwargs) y get_previous_by_FOO(**kwargs) . . . 293
C.10.3. get_FOO_filename() . . . 294
C.10.4. get_FOO_url() . . . 294

C.10.5. get_FOO_size() . . . 294
C.10.6. save_FOO_file(filename, raw_contents) . . . 294
C.10.7. get_FOO_height() and get_FOO_width() . . . 294
C.11. Atajos (Shortcuts) . . . 294
C.11.1. get_object_or_404() . . . 294
C.11.2. get_list_or_404() . . . 295
C.12. Utilizando SQL Crudo . . . 295

D. Referencia de las vistas genéricas **297**
D.1. Argumentos comunes a todas las vistas genéricas . . . 297
D.2. Vistas genéricas simples . . . 298
D.2.1. Representar una plantilla . . . 298
D.2.2. Redirigir a otra URL . . . 298
D.3. Vistas de listado/detalle . . . 299
D.3.1. Listas de objetos . . . 299
D.3.2. Vista de detalle . . . 300
D.4. Vistas genéricas basadas en fechas . . . 302
D.4.1. Índice de archivo . . . 302
D.4.2. Archivos anuales . . . 303
D.4.3. Archivos mensuales . . . 304
D.4.4. Archivos semanales . . . 306
D.4.5. Archivos diarios . . . 307
D.4.6. Archivo para hoy . . . 308
D.4.7. Páginas de detalle basadas en fecha . . . 308
D.5. Vistas genericas para Crear/Modificar/Borrar . . . 309
D.5.1. Vista de creación de objetos . . . 310
D.5.2. Vista de modificación de objetos . . . 311
D.5.3. Vista de borrado de objetos . . . 312

E. Variables de configuración **313**
E.1. Qué es un archivo de configuración . . . 313
E.1.1. Valores por omisión . . . 313
E.1.2. Viendo cuáles variables de configuración has cambiado . . . 313
E.1.3. Usando variables de configuración en código Python . . . 314
E.1.4. Modificando variables de configuración en tiempo de ejecución . . . 314
E.1.5. Seguridad . . . 314
E.1.6. Creando tus propias variables de configuración . . . 314
E.2. Indicando la configuración: DJANGO_SETTINGS_MODULE . . . 314
E.2.1. La utilidad django-admin.py . . . 315
E.2.2. En el servidor (mod_python) . . . 315
E.3. Usando variables de configuración sin fijar DJANGO_SETTINGS_MODULE . . . 315
E.3.1. Variables de configuración por omisión personalizados . . . 316
E.3.2. Es necesario que uses configure() o DJANGO_SETTINGS_MODULE . . . 316
E.4. Variables de configuración disponibles . . . 316
E.4.1. ABSOLUTE_URL_OVERRIDES . . . 316
E.4.2. ADMIN_FOR . . . 316
E.4.3. ADMIN_MEDIA_PREFIX . . . 317
E.4.4. ADMINS . . . 317
E.4.5. ALLOWED_INCLUDE_ROOTS . . . 317
E.4.6. APPEND_SLASH . . . 317
E.4.7. CACHE_BACKEND . . . 317
E.4.8. CACHE_MIDDLEWARE_KEY_PREFIX . . . 317
E.4.9. DATABASE_ENGINE . . . 317
E.4.10. DATABASE_HOST . . . 317
E.4.11. DATABASE_NAME . . . 318
E.4.12. DATABASE_OPTIONS . . . 318
E.4.13. DATABASE_PASSWORD . . . 318
E.4.14. DATABASE_PORT . . . 318
E.4.15. DATABASE_USER . . . 318

E.4.16. DATE_FORMAT . . . 318
E.4.17. DATETIME_FORMAT . . . 318
E.4.18. DEBUG . . . 318
E.4.19. DEFAULT_CHARSET . . . 319
E.4.20. DEFAULT_CONTENT_TYPE . . . 319
E.4.21. DEFAULT_FROM_EMAIL . . . 319
E.4.22. DISALLOWED_USER_AGENTS . . . 319
E.4.23. EMAIL_HOST . . . 319
E.4.24. EMAIL_HOST_PASSWORD . . . 319
E.4.25. EMAIL_HOST_USER . . . 319
E.4.26. EMAIL_PORT . . . 319
E.4.27. EMAIL_SUBJECT_PREFIX . . . 319
E.4.28. FIXTURE_DIRS . . . 320
E.4.29. IGNORABLE_404_ENDS . . . 320
E.4.30. IGNORABLE_404_STARTS . . . 320
E.4.31. INSTALLED_APPS . . . 320
E.4.32. INTERNAL_IPS . . . 320
E.4.33. JING_PATH . . . 320
E.4.34. LANGUAGE_CODE . . . 320
E.4.35. LANGUAGES . . . 320
E.4.36. MANAGERS . . . 321
E.4.37. MEDIA_ROOT . . . 321
E.4.38. MEDIA_URL . . . 321
E.4.39. MIDDLEWARE_CLASSES . . . 321
E.4.40. MONTH_DAY_FORMAT . . . 321
E.4.41. PREPEND_WWW . . . 321
E.4.42. PROFANITIES_LIST . . . 322
E.4.43. ROOT_URLCONF . . . 322
E.4.44. SECRET_KEY . . . 322
E.4.45. SEND_BROKEN_LINK_EMAILS . . . 322
E.4.46. SERIALIZATION_MODULES . . . 322
E.4.47. SERVER_EMAIL . . . 322
E.4.48. SESSION_COOKIE_AGE . . . 322
E.4.49. SESSION_COOKIE_DOMAIN . . . 322
E.4.50. SESSION_COOKIE_NAME . . . 322
E.4.51. SESSION_COOKIE_SECURE . . . 323
E.4.52. SESSION_EXPIRE_AT_BROWSER_CLOSE . . . 323
E.4.53. SESSION_SAVE_EVERY_REQUEST . . . 323
E.4.54. SITE_ID . . . 323
E.4.55. TEMPLATE_CONTEXT_PROCESSORS . . . 323
E.4.56. TEMPLATE_DEBUG . . . 323
E.4.57. TEMPLATE_DIRS . . . 323
E.4.58. TEMPLATE_LOADERS . . . 323
E.4.59. TEMPLATE_STRING_IF_INVALID . . . 324
E.4.60. TEST_RUNNER . . . 324
E.4.61. TEST_DATABASE_NAME . . . 324
E.4.62. TIME_FORMAT . . . 324
E.4.63. TIME_ZONE . . . 324
E.4.64. URL_VALIDATOR_USER_AGENT . . . 324
E.4.65. USE_ETAGS . . . 324
E.4.66. USE_I18N . . . 325
E.4.67. YEAR_MONTH_FORMAT . . . 325

F. Etiquetas de plantilla y filtros predefinidos **327**
F.1. Etiquetas predefinidas . . . 327
F.1.1. block . . . 327
F.1.2. comment . . . 327
F.1.3. cycle . . . 327
F.1.4. debug . . . 328
F.1.5. extends . . . 328
F.1.6. filter . . . 328
F.1.7. firstof . . . 328
F.1.8. for . . . 328
F.1.9. if . . . 329
F.1.10. ifchanged . . . 330
F.1.11. ifequal . . . 330
F.1.12. ifnotequal . . . 330
F.1.13. include . . . 331
F.1.14. load . . . 331
F.1.15. now . . . 331
F.1.16. regroup . . . 333
F.1.17. spaceless . . . 333
F.1.18. ssi . . . 334
F.1.19. templatetag . . . 334
F.1.20. url . . . 334
F.1.21. widthratio . . . 335
F.2. Filtros predefinidos . . . 335
F.2.1. add . . . 335
F.2.2. addslashes . . . 335
F.2.3. capfirst . . . 335
F.2.4. center . . . 335
F.2.5. cut . . . 335
F.2.6. date . . . 336
F.2.7. default . . . 336
F.2.8. default_if_none . . . 336
F.2.9. dictsort . . . 336
F.2.10. dictsortreversed . . . 336
F.2.11. divisibleby . . . 336
F.2.12. escape . . . 336
F.2.13. filesizeformat . . . 337
F.2.14. first . . . 337
F.2.15. fix_ampersands . . . 337
F.2.16. floatformat . . . 337
F.2.17. get_digit . . . 337
F.2.18. join . . . 338
F.2.19. length . . . 338
F.2.20. length_is . . . 338
F.2.21. linebreaks . . . 338
F.2.22. linebreaksbr . . . 338
F.2.23. linenumbers . . . 338
F.2.24. ljust . . . 338
F.2.25. lower . . . 338
F.2.26. make_list . . . 339
F.2.27. phone2numeric . . . 339
F.2.28. pluralize . . . 339
F.2.29. pprint . . . 339
F.2.30. random . . . 339
F.2.31. removetags . . . 340
F.2.32. rjust . . . 340
F.2.33. slice . . . 340
F.2.34. slugify . . . 340

F.2.35. stringformat . . . 340
F.2.36. striptags . . . 340
F.2.37. time . . . 340
F.2.38. timesince . . . 341
F.2.39. timeuntil . . . 341
F.2.40. title . . . 341
F.2.41. truncatewords . . . 341
F.2.42. truncatewords_html . . . 341
F.2.43. unordered_list . . . 341
F.2.44. upper . . . 342
F.2.45. urlencode . . . 342
F.2.46. urlize . . . 342
F.2.47. urlizetrunc . . . 342
F.2.48. wordcount . . . 342
F.2.49. wordwrap . . . 342
F.2.50. yesno . . . 343

G. El utilitario django-admin **345**
G.1. Uso . . . 345
G.2. Acciones Disponibles . . . 345
G.2.1. adminindex [appname appname ...] . . . 346
G.2.2. createcachetable [tablename] . . . 346
G.2.3. dbshell . . . 346
G.2.4. diffsettings . . . 346
G.2.5. dumpdata [appname appname ...] . . . 346
G.2.6. flush . . . 346
G.2.7. inspectdb . . . 346
G.2.8. loaddata [fixture fixture ...] . . . 347
G.2.9. reset [appname appname ...] . . . 348
G.2.10. runfcgi [options] . . . 348
G.2.11. runserver [número de puerto opcional, or direcciónIP:puerto] . . . 348
G.2.12. shell . . . 348
G.2.13. sql [appname appname ...] . . . 349
G.2.14. sqlall [appname appname ...] . . . 349
G.2.15. sqlclear [appname appname ...] . . . 349
G.2.16. sqlcustom [appname appname ...] . . . 349
G.2.17. sqlindexes [appname appname ...] . . . 349
G.2.18. sqlreset [appname appname ...] . . . 349
G.2.19. sqlsequencereset [appname appname ...] . . . 349
G.2.20. startapp [appname] . . . 349
G.2.21. startproject [projectname] . . . 349
G.2.22. syncdb . . . 349
G.2.23. test . . . 350
G.2.24. validate . . . 350
G.3. Opciones Disponibles . . . 350
G.3.1. --settings . . . 350
G.3.2. --pythonpath . . . 350
G.3.3. --format . . . 350
G.3.4. --help . . . 350
G.3.5. --indent . . . 350
G.3.6. --noinput . . . 351
G.3.7. --noreload . . . 351
G.3.8. --version . . . 351
G.3.9. --verbosity . . . 351
G.3.10. --adminmedia . . . 351

H. Objetos Petición y Respuesta **353**
H.1. HttpRequest . . . 353
H.1.1. Objetos QueryDict . . . 355
H.1.2. Un ejemplo completo . . . 356
H.2. HttpResponse . . . 357
H.2.1. Construcción de HttpResponses . . . 357
H.2.2. Establecer las cabeceras . . . 357
H.2.3. Subclases de HttpResponse . . . 358
H.2.4. Retornar Errores . . . 358
H.2.5. Personalizar la Vista 404 (Not Found) . . . 359
H.2.6. Personalizar la Vista 500 (Server Error) . . . 359

I. Docutils System Messages **361**

Preliminares

Reconocimientos

El aspecto más gratificante de trabajar con Django es la comunidad. Hemos sido especialmente afortunados de que Django haya atraído a tanta gente inteligente, motivada y amistosa. Un segmento 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 Beattie, Rod Begbie, Paul Bissex, Matt Boersma, Robbin Bonthond, 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 de ayudarnos.

Un especial agradecimiento a Simon Willison por escribir el capítulo de procesamiento de formularios. Realmente apreciamos la ayuda y nos enorgullece que la excelente redacción de Simon pueda ser parte de este libro.

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 on line; 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.

Sobre los autores

Adrian Holovaty, desarrollador Web y periodista, es uno de los creadores y desarrolladores del núcleo de Django. Es el fundador de EveryBlock, una Web startup local de noticias. Cuando no está trabajando en mejoras para Django, Adrian hackea en proyectos de beneficio público, como chicagocrime.org, uno de los mashups originales de Google Maps. Vive en Chicago y mantiene un weblog en holovaty.com.

Jacob Kaplan-Moss es uno de los principales desarrolladores de Django. En su empleo diurno, es el desarrollador principal para el Lawrence Journal-World, un periódico de dueños locales en Lawrence, Kansas, donde Django fue desarrollado. En el Journal-World, supervisa el desarrollo de Ellington, una plataforma de publicación online de noticias para compañías de medios de comunicación. A Jacob se lo puede encontrar online en jacobian.org.

Sobre el editor técnico

Jeremy Dunck es el principal desarrollador de Pegasus News, un sitio local personalizado con base en Dallas, Texas. Es uno de los primeros colaboradores de Greasemonkey y Django y ve la tecnología como una herramienta para la comunicación y el acceso al conocimiento.

Sobre los traductores

La traducción al español de *El libro de Django* fue posible gracias a la colaboración voluntaria de la comunidad Django en Español y Python Argentina. El proyecto se lleva a cabo desde `http://humitos.homelinux.net/django-book`. A la fecha, han contribuido de una u otra manera a este trabajo:

- Manuel Kaufmann <humitos en gmail.com>
- Martín Gaitán <gaitan en gmail.com>
- Leonardo Gastón De Luca <lgdeluca84 en gmail.com>
- Guillermo Heizenreder <gheize en gmail.com>
- Alejandro Autalán <alejandroautalan en gmail.com>
- Renzo Carbonara <gnuk0001 en gmail.com>
- Milton Mazzarri <milmazz en gmail.com>
- Ramiro Morales <ramiro+djbook en rmorales dot net>
- Juan Ignacio Rodríguez de León <euribates+django en gmail punto com>
- Percy Pérez Pinedo <percyp3 en gmail punto com>
- Tomás Casquero <tcasquero en gmail.com>
- Marcos Agustín Lewis <marcoslewis en gmail punto com>
- Leónidas Hernán Olivera <lholivera en gmail punto com>
- Federico M. Peretti <federico en perettifederico.com.ar>
- César Ballardini <cesar en ballardini.com.ar>
- Anthony Lenton <anthony en except.com.ar>
- César Roldán <cesar en hugoroldan.com.ar>
- Gonzalo Delgado <gonzalodel en gmail.com>

Sobre el libro

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

Hemos lanzado este libro 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 desde material técnico bien escrito, así que nosotros intentamos escribir una guía destacada que sirva además como referencia para 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" -- 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.

Introducción

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

Como los sitios web crecieron y se hicieron más ambiciosos, rápidamente se hizo obvio 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 Mosaic, el primer navegador web gráfico, fue desarrollado) 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 la web para siempre.

Ahora es duro 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.

Django fue inventado 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 de patrones comunes del desarrollo web, atajos para tareas frecuentes de programación y claras convenciones sobre cómo resolver problemas. Al mismo tiempo, Django intenta mantenerse fuera de tu camino, dejando que trabajes fuera del alcance del framework cuando es 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 un 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 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.

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

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 divertida -- el quid 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. 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 poderosos de forma rápida, con código limpio y de fácil mantenimiento.

En este capítulo ofrecemos una visión general de Django.

1.1. ¿Qué es un Framework Web?

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

Para contestar esa pregunta, consideremos el diseño de una aplicación Web escrita usando el estándar Common Gateway Interface (CGI), una forma popular de escribir aplicaciones Web alrededor del año 1998. En esa época, cuando escribías una aplicación CGI, hacías todo por ti mismo -- el equivalente a hacer una torta desde cero --. Por ejemplo, aquí hay un script CGI sencillo, escrito en Python, que muestra los diez libros más recientemente publicados de una base de datos:

```
#!/usr/bin/python

import MySQLdb

print "Content-Type: text/html"
print
print "<html><head><title>Libros</title></head>"
print "<body>"
print "<h1>Los ultimos 10 libros</h1>"
print "<ul>"

conexion = MySQLdb.connect(user='yo', passwd='dejame_entrar', db='mi_base')
cursor = conexion.cursor()
cursor.execute("SELECT nombre FROM libros ORDER BY fecha_pub DESC LIMIT 10")
for fila in cursor.fetchall():
    print "<li>%s</li>"% fila[0]

print "</ul>"
```

```
print "</body></html>"

conexion.close()
```

Este código es fácil de entender. Primero imprime una línea de "Content-Type", seguido de una línea en blanco, tal como requiere CGI. Imprime HTML introductorio, se conecta a la base de datos y ejecuta una consulta que obtiene los diez libros más recientes. 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 única página 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 16 líneas de Python y entender todo lo que hace, de principio a fin --. No hay más nada que aprender; no hay más código para leer. También es sencillo de utilizar: tan sólo guarda este código en un archivo llamado `ultimoslibros.cgi`, sube ese archivo a un servidor Web y visita esa página con un navegador.

Pero 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.

1.2. El patrón de diseño MVC

Comencemos con un rápido ejemplo 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 (las tablas de la base de datos)

from django.db import models

class Book(models.Model):
    name = models.CharField(maxlength=50)
    pub_date = models.DateField()


# views.py (la parte lógica)

from django.shortcuts import render_to_response
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})
```

```
# urls.py (la configuración URL)

from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'latest/$', views.latest_books),
)


# latest_books.html (la plantilla)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

Todavía no es necesario preocuparse por los detalles de *cómo* funciona esto -- tan sólo queremos que te acostumbres al diseño general --. Lo que hay que notar principalmente en este caso 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 código Python sencillo en lugar de escribir declaraciones SQL repetitivas.
- El archivo `views.py` contiene la lógica de la página, en la función `latest_books()`. A esta función se la denomina vista.
- El archivo `urls.py` especifica qué vista es llamada según el patrón URL. En este caso, la URL `/latest/` será manejada por la función `latest_books()`.
- El archivo `latest_books.html` es una plantilla HTML que describe el diseño de la página.

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 (N. de T.: por *loosely coupled*) 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 Capítulo 3 trata sobre las vistas, el Capítulo 4 sobre las plantillas, y el Capítulo 5 sobre los modelos. El Capítulo 5 profundiza también en la filosofía MVC de Django.

1.3. La historia de Django

Antes de continuar con más código, deberíamos tomarnos un momento para explicar la historia de Django. 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 acaba de inventar un framework.

Así es precisamente como fue creado 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 boreal de 2005, luego de haber desarrollado este framework hasta el punto en que estaba haciendo funcionar la mayoría de los sitios 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.

A pesar de que Django ahora es un proyecto de código abierto con colaboradores por todo el mundo, los desarrolladores originales de World Online todavía aportan una guía centralizada para el crecimiento del framework, y World Online colabora con otros aspectos importantes tales como tiempo de trabajo, materiales de marketing, y hosting/ancho de banda para el Web site del framework (`http://www.djangoproject.com/`).

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 admin, 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 diferencia entre ser *particularmente efectivo* para algo y *no ser 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 activamente mejorado casi diariamente. Los desarrolladores del framework tienen un alto grado de interés en asegurarse de que Django les ahorre tiempo a los desarrolladores, produzca aplicaciones que son fáciles de mantener y rindan bajo mucha carga. Aunque existan otras razones, los desarrolladores están motivados por sus propios deseos egoístas de ahorrarse tiempo a ellos mismos y disfrutar de sus trabajos. (Para decirlo sin vueltas, se comen su propia comida para perros).

1.4. Cómo leer este libro

Al escribir este libro, tratamos de alcanzar un balance entre legibilidad y referencia, con una tendencia a la legibilidad. Nuestro objetivo con este libro, 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 eso en mente, te recomendamos que leas los capítulos del 1 al 7 en orden. Ellos forman los fundamentos de cómo se usa Django; una vez que los hayas leído, serás capaz de construir sitios Web que funcionan sobre Django. Los capítulos restantes, los cuales se enfocan en características específicas de Django, pueden ser leídos en cualquier orden.

Los apéndices son para referencia. Ellos, junto con la documentación libre en `http://www.djangoproject.com/`, son probablemente lo que releerás de vez en cuando para recordar la sintaxis o buscar un resumen rápido de lo que hacen ciertas partes de Django.

1.4.1. Conocimientos de programación requeridos

Los lectores de este libro deben comprender las bases de la programación orientada a objetos e imperativa: estructuras de control (`if`, `while` y `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.

1.4.2. 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 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.

1.4.3. Nuevas características de Django

Tal como hicimos notar 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 escribimos.

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 antes mencionado y también revisa la documentación oficial de Django.

1.4.4. Obteniendo ayuda

Uno de los mayores beneficios de Django es su comunidad de usuarios amable y servicial. Para 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 en inglés de usuarios de Django se juntan miles de usuarios para preguntar y responder dudas. Suscríbete gratuitamente en `http://www.djangoproject.com/r/django-users` (inglés) o `http://groups.google.es/group/django-es` (español).
- 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.

1.5. ¿Qué sigue?

En el Capítulo 2, a continuación, empezaremos con Django, explicando su instalación y configuración inicial.

Capítulo 2

Empezando

Creemos que es mejor empezar con fuerza. En los capítulos que le siguen a este descubrirás los detalles y el alcance del framework Django, pero por ahora, confía en nosotros, este capítulo es divertido.

Instalar Django es fácil. Django se puede usar en cualquier sistema que corra Python, por eso es posible instalarlo de varias maneras. En este capítulo explicamos las situaciones más comunes de instalación de Django. El Capítulo 20 explica como utilizar Django en producción.

2.1. Instalar Python

Django está escrito 100 % en puro código Python, así que necesitarás instalar Python en tu computadora. Django necesita Python 2.3 o superior.

Si estás usando Linux o Mac OS X probablemente ya tienes Python instalado. Escribe `python` en una terminal. Si ves algo así, Python está instalado:

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Si ves un error como: `"command not found"` u `"orden no encontrada"`, tienes que bajar e instalar Python. Fíjate en http://www.python.org/download/ para empezar. La instalación es rápida y fácil.

2.2. Instalar Django

En esta sección explicamos dos opciones de instalación: instalar un lanzamiento oficial e instalar desde Subversion.

2.2.1. Instalar un lanzamiento oficial

La mayoría de la gente querrá instalar el lanzamiento oficial más reciente de http://www.djangoproject.com/download/. Django usa el método `distutils` estándar de instalación de Python, que en el mundo de Linux es así:

1. Baja el tarball, que se llamará algo así como *Django-0.96.tar.gz*
2. `tar xzvf Django-*.tar.gz`
3. `cd Django-*`
4. `sudo python setup.py install`

En Windows, recomendamos usar 7-Zip para manejar archivos comprimidos de todo tipo, incluyendo `.tar.gz`. Puedes bajar 7-Zip de http://www.djangoproject.com/r/7zip/.

Cambia a algún otro directorio e inicia `python`. Si todo está funcionando bien, deberías poder importar el módulo `django`:

```
>>> import django
>>> django.VERSION
(0, 96, None)
```

> **Nota**
>
> 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` 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.

2.2.2. Instalar Django desde Subversion

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 de Subversion.

Subversion es libre, es un sistema de control de versiones de código abierto similar a CVS, y es el que el equipo de Django utiliza para administrar cambios en el código base de Django. Puedes utilizar un cliente de Subversion para hacerte con el código fuente más actual de Django y, en cualquier momento, puedes actualizar tu copia local del código de Django, conocido como un *checkout local*, para obtener los últimos cambios y mejoras hechas por los desarrolladores de Django.

Al último código de desarrollo de Django se hace referencia como el *trunk*. El equipo de Django ejecuta sitios de producción sobre el trunk y procura permanecer estable.

Para obtener el trunk de Django, sigue los siguientes pasos:

1. Asegúrate de tener un cliente de Subversion instalado. Puedes conseguir este programa libremente desde http://subversion.tigris.org/, y puedes encontrar documentación excelente en http://svnbook.red-bean.com/.
2. Haz un check out del trunk usando el comando `svn co http://code.djangoproject.com/svn/django/trunk djtrunk`.
3. Crea `site-packages/django.pth` y agrega el directorio `djtrunk` a este, o actualiza tu `PYTHONPATH` agregando `djtrunk`.
4. Incluye `djtrunk/django/bin` en el PATH de tu sistema. Este directorio incluye utilidades de administración como `django-admin.py`.

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

Luego de descargarlo desde Subversion y haber seguido los pasos anteriores, no necesitas ejecutar `python setup.py install` --¡Acabas de hacer este trabajo a mano!

Debido a que el trunk de Django cambia a menudo corrigiendo bugs y agregando funcionalidades, probablemente quieras actualizarlo con frecuencia -- a cada hora, si eres un obsesivo. Para actualizar el código, solo ejecuta el comando `svn update` desde el directorio `djtrunk`. Cuando ejecutes este comando, Subversion contactará http://code.djangoproject.com, 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.

2.3. Configurando la base de datos

El único prerequisito 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 de base de datos*, para esto 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 "Comenzando un proyecto" -- pero creénos, 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 tres motores de base de datos:

- PostgreSQL (http://www.postgresql.org/)
- SQLite 3 (http://www.sqlite.org/)
- MySQL (http://www.mysql.com/)

Se está trabajando para admitir Microsoft SQL Server y Oracle. El sitio web de Django siempre contendrá la última información acerca de las base de datos admitidas.

A nosotros el que más nos gusta es PostgreSQL, por razones que exceden el alcance de este libro, por eso lo mencionamos primero. Sin embargo, todos los motores que listamos aquí trabajan bien con Django.

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 2.5.

En Windows, obtener los drivers binarios de la base de datos es a veces un proceso complicado. Ya que sólo estás iniciándote en Django, recomendamos usar Python 2.5 y el soporte incluido para SQLite. La compilación de drivers puede ser estresante.

2.3.1. Usar Django con PostgreSQL

Si estás utilizando PostgreSQL, necesitarás el paquete `psycopg` disponible en http://www.djangoproject.com/r/python-pgsql/. Toma nota de la versión que estás usando (1 o 2); necesitarás esta información luego.

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

2.3.2. Usar Django con SQLite 3

Si estás usando una versión de Python igual o posterior a 2.5, ya tienes SQLite. Si estás trabajando con Python 2.4 o menor, necesitas SQLite 3 --no la versión 2-- desde http://www.djangoproject.com/r/sqlite/ y el paquete `pysqlite` desde http://www.djangoproject.com/r/python-sqlite/. Asegúrate de tener `pysqlite` en una versión 2.0.3 o superior.

En Windows, puedes omitir la instalación separada de los binarios de SQLite, ya que están enlazados dentro de los binarios de `pysqlite`.

2.3.3. 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` desde http://www.djangoproject.com/r/python-mysql/.

2.3.4. 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).

2.4. Comenzando 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 iniciales. Crea un nuevo directorio para empezar a trabajar, quizás algo como `/home/username/djcode/`, e ingresa a este directorio.

> **Nota**
>
> `django-admin.py` debería estar en el PATH de tu sistema si instalaste Django con la utilidad `setup.py`. Si hiciste un check out desde Subversion, puedes encontrarlo en `djtrunk/django/bin`. Como vas a utilizar con frecuencia `django-admin.py`, considera agregarlo a tu PATH. En Unix, puedes hacer un link simbólico de `/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, necesitarás actualizar tu variable de entorno `PATH` .

Ejecuta el comando `django-admin.py startproject mysite` para crear el directorio `mysite` en el directorio actual.

Echemos un vistazo a lo que `startproject` creó:

```
mysite/
    __init__.py
    manage.py
    settings.py
    urls.py
```

Estos archivos son los siguientes:

- `__init__.py`: Un archivo requerido para que Python trate a este directorio como un paquete (i.e. un grupo de módulos).
- `manage.py`: Una utilidad de línea de comandos que te deja interactuar con este proyecto de Django de varias formas.
- `settings.py`: Opciones/configuraciones para este proyecto de Django.
- `urls.py`: La declaración de las URL para este proyecto de Django; una "tabla de contenidos" de tu sitio hecho con Django.

> **¿Dónde debería estar este directorio?**
>
> Si vienes de 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.

2.4.1. El servidor de desarrollo

Django incluye un servidor web ligero 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 de producción (i.e., Apache) hasta que estés listo para la 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.

Entra en el directorio `mysite`, si aún no lo has hecho, y ejecuta el comando `python manage.py runserver`. Verás algo parecido a esto:

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

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 Capítulo 20 para información sobre cómo hacerlo con Django.

> **Cambiar el host o 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, pasa este como un argumento de línea de comandos:
>
> ```
> python manage.py runserver 8080
> ```
>
> También puedes cambiar las direcciones de IP que escucha el servidor. Esto es utilizado especialmente si quieres compartir el desarrollo de un sitio con otros desarrolladores. Lo siguiente:
>
> ```
> python manage.py runserver 0.0.0.0:8080
> ```
>
> hará que Django escuche sobre cualquier interfaz de red, permitiendo que los demás equipos puedan conectarse al servidor de desarrollo.

Ahora que el servidor está corriendo, visita `http://127.0.0.1:8000/` con tu navegador web. Verás una página de "Bienvenido a Django" sombreada con un azul pastel agradable. ¡Funciona!

2.5. ¿Qué sigue?

Ahora que tienes todo instalado y el servidor de desarrollo corriendo, en el **‘próximo capítulo‘_** escribirás algo de código básico que muestra cómo servir páginas Web usando Django.

Duplicate explicit target name: “próximo capítulo”.

Capítulo 3

Los principios 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 de Django. Por supuesto, el sitio no hace nada útil todavía -- sólo muestra el mensaje "It worked!". Cambiemos eso. Este capítulo presenta cómo crear paginas web dinámicas con Django.

3.1. Tu primera Vista: Contenido dinámico

Lo primero que haremos es crear 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 -- al contrario, los contenidos cambian de acuerdo con el resultado de un cálculo (en este caso, el cálculo de la hora actual). Este simple ejemplo no involucra una base de datos cualquier tipo de entrada del usuario, sólo muestra la salida del reloj interno del servidor.

Para crear esta página, crearemos una *función de vista*. Una función de vista, o *vista* en pocas palabras, es una simple función de Python que toma como argumento una petición Web y retorna una respuesta Web. Esta respuesta puede ser el contenido HTML de la página web, una redirección, o un error 404, o un documento XML, o una imagen... o en realidad, cualquier cosa. La vista en sí contiene toda la lógica necesaria para retornar esa respuesta. El código puede encontrarse donde quieras, mientras que se encuentre dentro de tu Python path. No hay otro requerimiento -- no hay "magia", por así decirlo. Por poner el código en *algún lugar*, creemos un archivo llamado `views.py` en el directorio `mysite`, el cual creamos en el capítulo anterior.

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

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Repasemos el código anterior línea a línea:

- Primero, importamos la clase `HttpResponse`, la cual pertenece al módulo `django.http`. Para ver más detalles de los objetos `HttpRequest` y `HttpResponse` puedes consultar el Apéndice H.
- Luego importamos el módulo `datetime` de la biblioteca estándar de Python, el conjunto de módulos útiles que vienen con Python. El módulo `datetime` contiene varias funciones y clases para trabajar con fechas y horas, incluyendo una función que retorna la hora actual.
- A continuación, definimos la función llamada `current_datetime`. Esta es una función de vista. Cada función de vista toma como primer argumento un objeto `HttpRequest`, al que típicamente se le asigna el nombre `request`.

  Nota que el nombre de la función de vista no importa; no tiene que ser nombrada de una determinada manera para que Django la reconozca. La llamamos `current_datetime` aquí, porque el nombre indica claramente lo que hace, pero se podría llamar `super_duper_awesome_current_time`, o algo más repugnante. A Django no le interesa. La siguiente sección explica cómo Django encuentra esta funcion.
- La primera línea de código dentro de la función calcula la fecha/hora actual, como un objeto `datetime.datetime`, y almacena el resultado en la variable local `now`.

- 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 porcentaje después de la cadena de caracteres, significa "Reemplaza el `%s` por el valor de la variable now.". (Sí, el HTML es inválido, pero estamos tratando de mantener el ejemplo simple y corto)
- Por último, la vista retorna un objeto `HttpResponse` que contiene la respuesta generada. Cada función de vista es responsable de retornar un objeto `HttpResponse`. (Hay excepciones, pero lo haremos más adelante)

Zona Horaria de Django

Django incluye una opción `TIME_ZONE` que por omisión es `America/Chicago`. Probablemente no es donde vivas, por lo que puedes cambiarlo en tu `settings.py`. Veáse el Apéndice E para más detalles.

3.2. Mapeando URLs a Vistas

Repasemos, esta función de vista retorna un página HTML que contiene la fecha y hora actual. ¿Pero cómo le decimos a Django que utilice ese código?. Ahí es donde vienen las *URLconfs*.

Una *URLconf* es como una tabla de contenido 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 URL, llama a este otro código". Recuerda que estas funciones de vista deben estar en tu Python path.

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/python2.4/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/python2.4/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 `ImportError`

Si estás interesado en ver el valor de tu Python path, abre un interprete interactivo de Python y escribe `import sys`, seguido de `print sys.path`.

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

Cuando ejecutaste `django-admin.py startproject` en el capítulo anterior, el script creó automáticamente una URLconf por tí: el archivo `urls.py`. Editemos ese archivo. Por omisión, se verá como:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.apps.foo.urls.foo')),

    # Uncomment this for admin:
#     (r'^admin/', include('django.contrib.admin.urls')),
)
```

Repasemos el código anterior línea a línea:

- La primera línea importa todos los objetos desde el módulo `django.conf.urls.defaults`, incluyendo una función llamada `patterns`.

- La segunda línea llama a la función `patterns()` y guarda el resultado en una variable llamada `urlpatterns`. La función `patterns()` sólo recibe un argumento -- la cadena de caracteres vacía. El resto de las líneas están comentadas. (La cadena de caracteres puede ser usada para proveer un prefijo común para las funciones de vista, pero dejemos este uso más avanzado para más adelante).

Lo principal que debemos 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 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 esta vacía, Django asume que acabas de crear el proyecto, por lo tanto, muestra ese mensaje).

Editemos este archivo para exponer nuestra vista `current_datetime`:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
)
```

Hicimos dos cambios aquí. Primero, importamos la vista `current_datetime` desde el módulo (`mysite/views.py`, que en la sintaxis de import de Python se traduce a `mysite.views`). Luego, agregamos la línea `(r'^time/$', current_datetime),`. Esta línea hace referencia a un *URLpattern* -- es 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 ese patrón.

En pocas palabras, le estamos diciendo a Django que cualquier petición a la URL `/time` será manejada por la función de vista `current_datetime`.

Algunas cosas que vale la pena resaltar:

- Notemos que, en este ejemplo, pasamos la función de vista `current_datetime` 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?
- La `r` en `r'^time/$'` significa que `'^time/$` 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 `'^time/$'` para que coincida con `/time/`. Django automáticamente agrega una barra antes de toda expresión. A primera vista esto parece raro, pero una URLconf puede ser incluida en otra URLconf, y el dejar la barra de lado simplifica mucho las cosas. Esto se retoma en el Capítulo 8.
- El caracter *acento circunflejo* (`^`) y el carácter *signo de dólar* (`$`) son importantes. 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 `'^time/'` (sin el signo de dólar al final), entonces *cualquier* URL que comience con `time/` concordaría, así como `/time/foo` y `/time/bar`, no sólo `/time/`. Del mismo modo, si dejamos de lado el carácter acento circunflejo inicial (`'time/$'`), Django concordaría con *cualquier* URL que termine con `time/`, así como `/foo/bar/time/`. Por lo tanto, usamos tanto el acento circunflejo como el signo de dólar para asegurarnos que sólo la URL `/time/` concuerde. Nada más y nada menos.

  Quizás te preguntes qué pasa si alguien intenta acceder a `/time`. Esto es manejado como esperarías (a través de un redireccionamiento) siempre y cuando `APPEND_SLASH` tenga asignado el valor `True`. (Veáse el Apéndice E para una buena lectura sobre este tema).

Para probar nuestro 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 ya lo tenías corriendo, está bien también. 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 `http://127.0.0.1:8000/time/`. Deberías ver la salida de tu vista de Django.

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

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)
[^/]+	Todos los caracteres excepto la barra.
*	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

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

3.3. Cómo procesa una petición Django

Debemos señalar varias cosas en lo que hemos visto. Este es el detalle de lo que sucede cuando ejecutas el servidor de desarrollo de Django y hacemos una petición a una página Web.

- El comando `python manage.py runserver` importa un archivo llamado `settings.py` desde el mismo directorio. Este archivo contiene todo tipo de configuraciones opcionales para esta instancia de Django en particular, pero 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 `urls.py`? Bueno, el `settings.py` generado automáticamente tenía un `ROOT_URLCONF` que apunta al `urls.py` generado automáticamente. ¡Qué conveniente!
- Cuando llega una petición-- digamos, una petición a la URL `/time/` -- 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` luego).
- La función de vista es responsable de retornar un objeto `HttpResponse`.

Conoces ahora lo básico sobre cómo hacer páginas Web con Django. Es muy sencillo, realmente -- sólo tenemos 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.

3.3.1. 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.

El flujo típico -- resolución de URLconf a una función de vista que retorna un `HttpResponse`-- puede ser cortocircuitado o ***augmented*** mediante middleware. Los secretos del middleware serán tratados en profundidad en el Capítulo 15, pero un esquema (ver Figura 3-1) te ayudará conceptualmente a poner todas las piezas juntas.

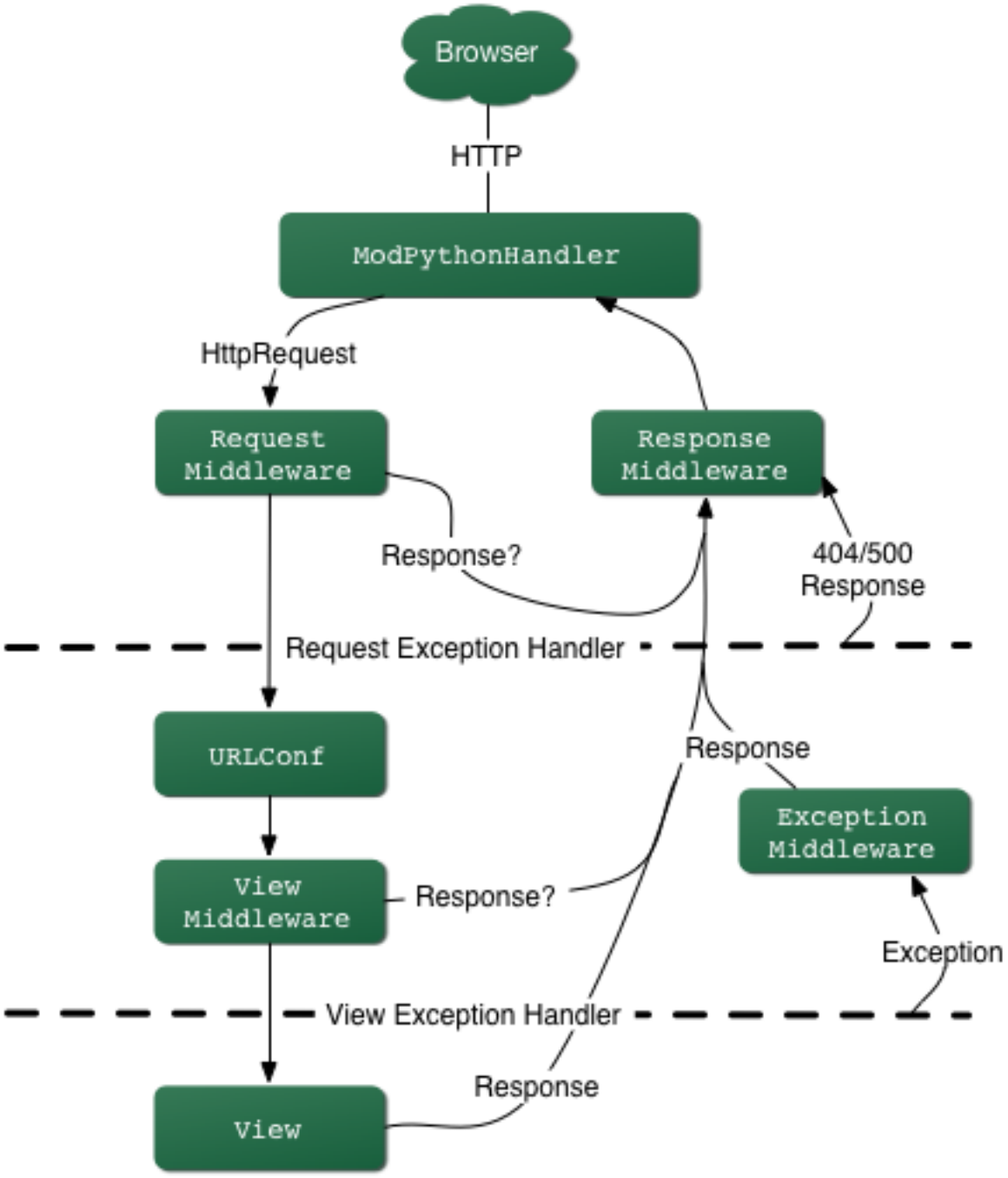

Figura 3.1: El flujo completo de un petición y una respuesta Django.

Cuando llega una petición HTTP desde el navegador, un *manejador* específico a cada servidor construye la `HttpRequest`, para pasarla a los componentes y maneja 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 ***augmenting*** los objetos `HttpRequest` así como también para proveer 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.

3.4. URLconfs y el acoplamiento débil

Ahora es el momento de resaltar una parte clave de filosofía detrás de las URLconf y detrás de Django en general: el principio de acoplamiento débil (*loose coupling*). Para explicarlo simplemente, 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 cuál nos mostraba la fecha y la hora actual. Si quieres cambiar la URL de tu aplicación -- digamos, mover desde `/time` a `/currenttime/` -- 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.

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

3.5. Errores 404

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

Para averiguarlo, prueba ejecutar el servidor de desarrollo Django e intenta acceder a una página Web como `http://127.0.0.1:8000/hello/` o `http://127.0.0.1:8000/does-not-exist/`, o mejor como `http://127.0.0.1:8000/` (la "raíz" del sitio). Deberías ver el mensaje "Page not found" (ver la Figura 3-2). (Es linda, ¿no? A la gente de Django seguro le gustan los colores pasteles). 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.

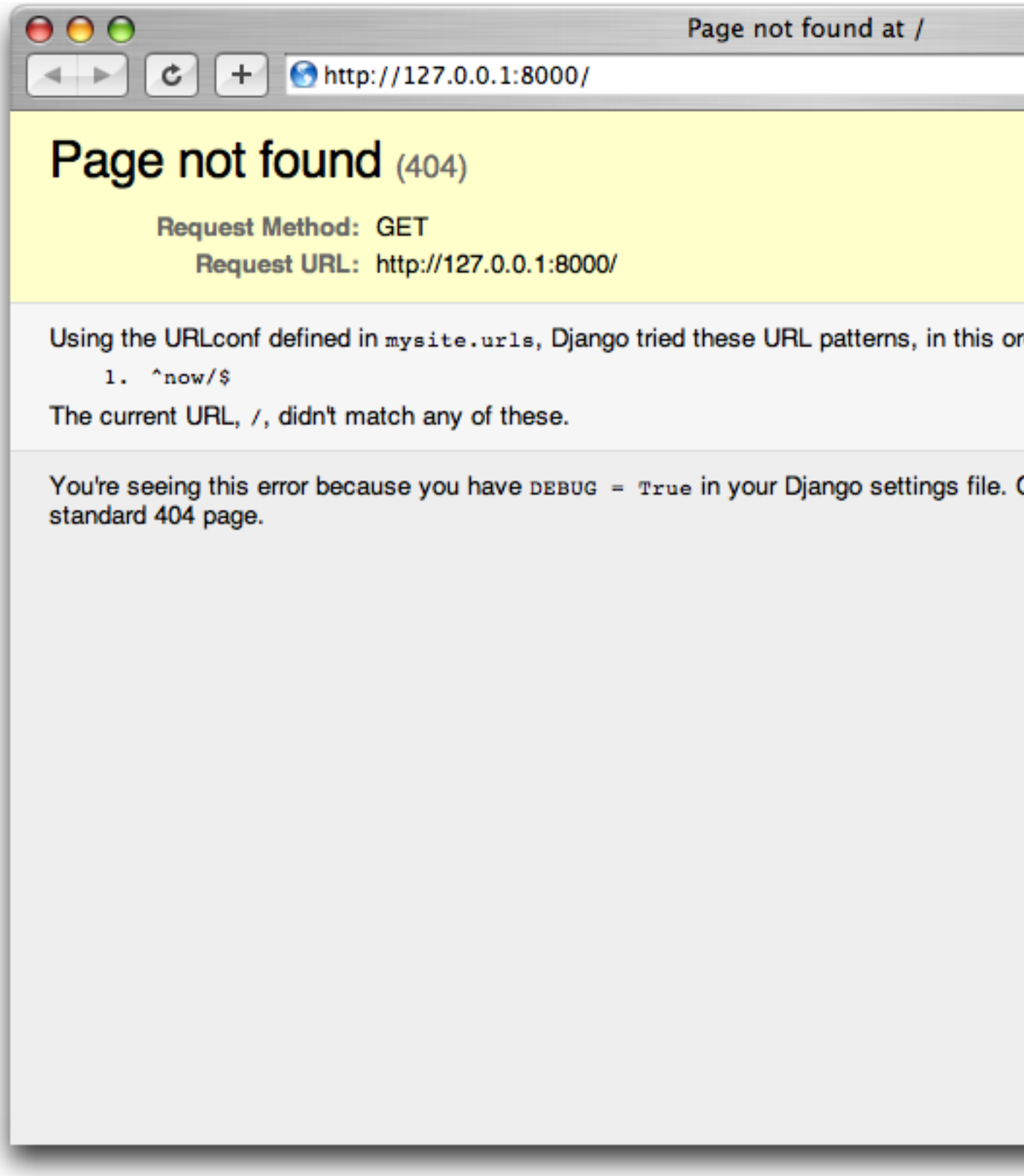

Figura 3.2: Página 404 de Django

3.6. Tu segunda Vista: URLs dinámicas

En la primer vista de ejemplo, el contenido de la página -- la fecha/hora actual -- eran dinámicas, pero la URL (`/time`) era estática. En la mayoría de las aplicaciones Web, sin embargo, la URL contiene parámetros que influyen en la salida de la página.

Vamos a crear una segunda 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 `/time/plus/1/` muestre la fecha/hora una hora más adelantada, la página `/time/plus/2/` muestre la fecha/hora dos horas más adelantada, la página `/time/plus/3/` muestre la fecha/hora tres horas más adelantada, y así.

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

```
urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/1/$', one_hour_ahead),
    (r'^time/plus/2/$', two_hours_ahead),
    (r'^time/plus/3/$', three_hours_ahead),
    (r'^time/plus/4//$', four_hours_ahead),
)
```

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 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.

3.6.1. 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 cadena de consulta!", algo como `/time/plus?hours=3`, en la cual la hora será designada por el parámetro `hours` 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 `/time/plus/3` es mucho más limpia, más simple, más legible, más fácil de dictarse a alguien y . . . justamente más bonita 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 bonitas, haciendo más fácil el usarlas que el no usarlas.

3.6.2. Comodines en los patrones URL

Continuando con nuestro ejemplo `hours_ahead`, pongámosle un comodín al patrón URL. Como ya se mencionó antes, un patrón URL es una expresión regular; de aquí, es que usamos el patrón de expresión regular `\d+` para que coincida con uno o más dígitos:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/\d+/$', hours_ahead),
)
```

Este patrón coincidirá con cualquier URL que sea como `/time/plus/2/`, `/time/plus/25/`, o también `/time/plus/100000000000`
Ahora que lo pienso, podemos limitar el lapso máximo de horas en 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}`:

```
(r'^time/plus/\d{1,2}/$', hours_ahead),
```

> **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 querramos guardar. En el caso del ejemplo, queremos guardar cualquier número que se anotará en la URL, entonces pongamos paréntesis alrededor de `\d{1,2}`:

```
(r'^time/plus/(\d{1,2})/$', hours_ahead),
```

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 la vista anterior `current_datetime`, nos quedará algo así:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

Con cuidado, vamos a escribir la vista `hours_ahead`.

> **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.

`hours_ahead` es muy similar a `current_datetime`, vista que escribimos antes, sólo que con una diferencia: tomará un argumento extra, el número de horas que debemos adelantar. Agrega al archivo `views.py` lo siguiente:

```
def hours_ahead(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Repasemos el código anterior línea a línea:

- Tal como hicimos en la vista `current_datetime`, importamos la clase `django.http.HttpResponse` y el módulo `datetime`.
- La función de vista `hours_ahead`, toma *dos* parámetros: `request` y `offset`.
  - `request` es un objeto `HttpRequest`, al igual que en `current_datetime`. 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 `/time/plus/3/`, entonces el `offset` debería ser la cadena de caracteres `'3'`. Si la petición URL fuera `/time/plus/21/`, entonces el `offset` debería ser la cadena de caracteres `'21'`. Notar 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'`.

    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`. Esto convierte el valor de la cadena de caracteres a entero.

  Tener 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'`. Sin embargo, en este ejemplo no debemos preocuparnos de atrapar la excepción, porque tenemos la certeza que la variable `offset` será una cadena de caracteres conformada sólo por dígitos. Sabemos esto, por el patrón URL de la expresión regular en el URLconf -- `(\d{1,2})`-- captura sólo dígitos. Esto ilustra otra ventaja de tener un URLconf: nos provee un primer nivel de validación de entrada.
- La siguiente línea de la función 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 `current_datetime`. Una pequeña diferencia en esta línea, 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 `current_datetime`.

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/time/plus/3/` para verificar que lo que hicimos funciona. Luego prueba `http://127.0.0.1:8000/time/plus/5/`. Para terminar visita `http://127.0.0.1:8000/time/plus/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 en la sección "Errores 404" anteriormente. La URL `http://127.0.0.1:8000/time/plus/` (*sin* horas designadas) debería también mostrar un error 404.

Si estás siguiendo el libro y programando al mismo tiempo, notarás que el archivo `views.py` ahora contiene dos vistas. (Omitimos la vista `current_datetime` del ejemplo anterior sólo por claridad). Poniéndolas juntas, veríamos algo similar a esto:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now%s.</body></html>"% now
    return HttpResponse(html)

def hours_ahead(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In%s hour(s), it will be%s.</body></html>"% (offset, dt)
    return HttpResponse(html)
```

3.7. 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 `hours_ahead`:

```
def hours_ahead(request, offset):
    #offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In%s hour(s), it will be%s.</body></html>"% (offset, dt)
    return HttpResponse(html)
```

Ejecuta el servidor de desarrollo y navega a `/time/plus/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: str"`.

¿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`. Es el típico pequeño *bug* que todo programador comete en algún momento.

El punto 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.

Aquí comentamos algunas cosas a destacar:

- 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), el archivo en el cuál la excepción fue lanzada, y el número de línea que contiene el error.
- Abajo de la información clave de la excepción, la página muestra el ***traceback*** de Python para dicha excepción. Este es el ***traceback*** estándar que se obtiene en el interprete de Python, sólo que más interactivo. Por cada marco de la 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 click en la línea de código (en gris oscuro) para ver varias líneas anteriores y posteriores a la línea errónea, lo que nos brinda un poco de contexto.

  Haz click en *"Locals vars"* debajo de cualquier marco de la pila para ver la tabla de todas las variables locales y sus valores, en ese marco y en la posición exacta de código en el cual fue lanzada la excepción. Esta información de depuración es invaluable.
- Nota el texto *"Switch to copy-and-paste view"* debajo de la cabecera *"Traceback"*. Haz click en esas palabras, y el ***traceback*** cambiará a una versión que te permitirá fácilmente copiar y pegar. Usando esto para cuando necesitemos compartir el traceback de la excepción con otros para obtener soporte técnico -- como los amables colegas que encontraremos en el canal de IRC o la lista de correo de Django.
- 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 como las cabeceras CGI. El "Apéndice H'_'_ es una completa referencia sobre toda la información que contienen los objetos peticiones.

  Inline literal start-string without end-string.

  Más abajo, la sección *"Settings"* lista la configuración de la instalación de Django en particular. El "Apéndice E'_'_, cubre en detalle los ajustes de configuración disponibles. Por ahora, sólo mira los ajustes para tener una idea de la información disponible.

  Inline literal start-string without end-string.

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 normalmente de nuevo.

¿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 la sentencia `print`. En cualquier punto de la vista, temporalmente podemos insertar un `assert False` para provocar una página de error. Luego, podremos ver las variables locales y el estado del programa. (Hay maneras más avanzadas de depurar las vista en Django, lo explicaremos más adelante, pero esto es la forma más rápida y fácil).

Finalmente, es obvio que la mayor parte de la información es delicada -- expone las entrañas del código fuente de Python, como también de 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.)

3.8. ¿Qué sigue?

Hasta ahora hemos producido las vistas mediante código HTML dentro del código Python. Desafortunadamente, esto es 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 capítulo'_**

Duplicate explicit target name: "próximo capítulo".

malformed hyperlink target.

malformed hyperlink target.

Explicit markup ends without a blank line; unexpected unindent.

Capítulo 4

El sistema de plantillas de Django

En el capítulo anterior, quizás notaste algo extraño en cómo retornábamos el texto en nuestras vistas de ejemplos. A saber, el HTML fue codificado [2] directamente en nuestro código Python.

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 subyacente código de Python, 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.

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

Vamos a sumergirnos con una simple plantilla de ejemplo. Esta plantilla describe una página HTML que agradece a una persona por un pedido de la empresa. Piensa en esto como un formulario de una carta:

```
<html>
<head><title>Ordering notice</title></head>

<body>

<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>

<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>
```

```
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

Esta plantilla es un HTML básico con algunas variables y etiquetas de plantillas agregadas. Vamos paso a paso a través de este:

- Cualquier texto encerrado por un par de llaves (por ej. `{{ person_name }}`) es una *variable*. Esto significa "insertar el valor de la variable con el nombre tomado". ¿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 ordered_warranty %}`) es una *etiqueta de plantilla*. La definición de etiqueta es bastante extensa: una etiqueta sólo le indica al sistema de plantilla "haz algo".

  Este ejemplo de plantilla contiene dos etiquetas: la etiqueta `{% for item in item_list %}` (una etiqueta `for`) y la etiqueta `{% if ordered_warranty %}` (una etiqueta `if`).

  Una etiqueta `for` actúa como un simple constructor de bucle, dejándote recorrer sobre cada uno de los items de una secuencia. Una etiqueta `if`, como quizás esperabas, actúa como una cláusula lógica "if". En este caso en particular, la etiqueta chequea cuando el valor de la variable `ordered_warranty` evalúa a `True`. Si lo hace, el sistema de plantillas mostrará todo entre `{% if ordered_warranty %}` y `{% endif %}`. Si no, el sistema de plantillas no mostrará esto. El sistema de plantillas también admite `{% else %}` y otras varias cláusulas lógicas.
- Finalmente, el segundo párrafo de esta plantilla tiene un ejemplo de un *filtro*, con el cual puedes alterar la exposición de una variable. En este ejemplo, `{{ ship_date|date:"F j, Y" }}`, estamos pasando la variable `ship_date` por el filtro `date`, tomando el filtro `date` el argumento `"F j, Y"`. El filtro `date` formatea fechas en el formato tomado, especificado por ese argumento. Los filtros se conceden mediante el uso de un caracter pipe (`|`), como una referencia a las tuberías de Unix.

Cada plantilla de Django tiene acceso a varias etiquetas y filtros incorporados, algunas de ellas serán tratadas en la sección que sigue. El Apéndice F contiene la lista completa de etiquetas y filtros, y ésta es una buena idea para familiarizarse con esta lista, de modo que sepas qué es posible. También es posible crear tus propios filtros y etiquetas, los cuales cubriremos en el **'Capítulo 10'_**.

4.2. Empleo del sistema de plantillas

Para usar el sistema de plantillas en el código Python, sólo sigue estos dos pasos:

1. Crea un objeto `Template` brindando el código crudo de la plantilla como una cadena. Django también ofrece un camino para crear objetos `Template` por la designación de una ruta al archivo de plantilla en el sistemas de archivo; vamos a examinar esto en un rato.
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.

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

4.2.1. Creación de objetos Template

La manera fácil de crear objetos `Template` es instanciarlo 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 como funciona este código.

Ejemplos del Intérprete Interactivo

Durante todo el libro, hemos expuesto sesiones de ejemplo del intérprete Python interactivo. Puedes reconocer estos ejemplos por el triple signo mayor-que (>>>), el cuál designa el prompt del intérprete. Si estás copiando los ejemplos del libro, no copies estos signos mayor-que.
Las sentencias multilíneas en el intérprete interactivo son rellenadas con tres puntos (...), por ejemplo:

```
>>> print """This is a
... string that spans
... three lines."""
This is a
string that spans
three lines.
>>> def my_function(value):
...     print value
>>> my_function('hello')
hello
```

Esos tres puntos al comienzo de una línea adicional son insertados por el shell de Python -- no son parte de nuestra entrada. Los incluimos aquí por ser fiel a la salida actual del intérprete. Si estás copiando nuestros ejemplos para seguirlos, no copies esos puntos.

Desde dentro del directorio del proyecto creado por `django-admin.py startproject` (como se expuso en el Capítulo 2), escribe `python manage.py shell` para comenzar el intérprete interactivo. Aquí hay una base:

```
>>> from django.template import Template
>>> t = Template("My name is {{ name }}.")
>>> 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 simple forma en que Python "identifica" un objeto de `Template`.

Variables de configuración de Django

Cuando usas Django, necesitas indicarle qué valores usar para sus variables de configuración. Interactivamente, suele usarse `python manage.py shell`, pero tienes otras opciones descriptas en el Apéndice E.

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

```
>>> from django.template import Template
>>> t = Template('{% notatag %} ')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ...
  django.template.TemplateSyntaxError: Invalid block tag: 'notatag'
```

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

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

4.2.2. Renderizar una plantilla

Una vez que tienes un objeto `Template`, le puedes pasar datos brindando un *contexto*. Un contexto es simplemente un conjunto de variables y sus valores asociados. Una plantilla usar estas variables para poblar 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 mapeando nombres de variables con valores. La llamada al método `render()` del objeto `Template` con el contexto "rellena" la plantilla:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ name }}.")
>>> c = Context({"name": "Stephane"})
>>> t.render(c)
'My name is Stephane.'
```

> **Diccionarios y Contextos**
>
> Un diccionario de Python es un mapeo entre llaves conocidas y valores de variables. Un `Context` es similar a un diccionario, pero un `Context` provee funcionalidad adicional, como se cubre en el **'Capítulo 10'** _.

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 sensible a mayúsculas-minúsculas.

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

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>Thanks for ordering {{ product }} from {{ company }}. It's scheduled
... to ship on {{ ship_date|date:"F j, Y" }}.</p>
...
... {% if ordered_warranty%}
... <p>Your warranty information will be included in the packaging.</p>
... {% endif%}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'product': 'Super Lawn Mower',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2009, 4, 2),
...     'ordered_warranty': True})
>>> t.render(c)
"<p>Dear John Smith,</p>\n\n<p>Thanks for ordering Super Lawn Mower from
Outdoor Equipment. It's scheduled \nto ship on April 2, 2009.</p>\n\n\n
<p>Your warranty information will be included in the packaging.</p>\n\n\n
<p>Sincerely,<br />Outdoor Equipment</p>"
```

Vamos paso a paso por este código de a una sentencia a la vez:

- Primero, importamos la clase `Template` y `Context`, ambas se encuentran en el módulo `django.template`.
- Guardamos el texto crudo de nuestra plantilla en la variable `raw_template`. Note que usamos triple comillas para designar la cadena de caracteres, debido a que envuelven varias líneas; en el código Python, las cadenas de caracteres designadas con una sola comilla marca que no puede envolver 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 nombres de variables con valores. Aquí, por ejemplo, especificamos que la `person_name` es `'John Smith'`, `product` es `'Super Lawn Mower'`, y así sucesivamente.
- Finalmente, llamamos al método `render()` sobre nuestro objeto de plantilla, pasando a este el contexto. Este retorna la plantilla renderizada -- esto es, reemplaza las variables de la plantilla con los valores actuales de las variables, y ejecuta cualquier bloque de etiquetas.

  Nota que el párrafo de garantía fue mostrado porque la variable `ordered_warranty` evaluó a `True`. También nota que la fecha `April 2, 2009`, es mostrada acorde al formato de cadena de caracteres `F j, Y`. (Explicaremos los formatos de cadenas de caracteres para el filtro `date` a la brevedad).

  Si eres nuevo en Python, quizás te asombre porqué 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 esta, 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 de usar el sistema de plantillas de Django: sólo escribe una plantilla, crea un objeto `Template`, crea un `Context`, y llama al método `render()`.

4.2.3. Múltiples contextos, mismas plantillas

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

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print t.render(Context({'name': 'John'}))
Hello, John
>>> print t.render(Context({'name': 'Julie'}))
Hello, Julie
>>> print t.render(Context({'name': 'Pat'}))
Hello, Pat
```

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 a `render()` sobre este muchas veces:

```
# Bad
for name in ('John', 'Julie', 'Pat'):
    t = Template('Hello, {{ name }}')
    print t.render(Context({'name': name}))

# Good
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
    print t.render(Context({'name': name}))
```

El análisis 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. En contraste con el motor de plantillas de XML, que incurren en la actividad general de un analizador XML, tienden a ser órdenes de magnitud más lentos que el motor de renderizado de Django.

4.2.4. Búsqueda del contexto de una variable

En los ejemplos hasta el momento, pasamos simples valores en 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 atravesar estructuras de datos complejas en las plantillas de Django ese el 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, supone que pasas un diccionario de Python a una plantilla. Para acceder al valor de ese diccionario por su clave, usa el punto:

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Sally is 43 years old.'
```

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('The month is {{ date.month }} and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
'The month is 5 and the year is 1993.'
```

Este ejemplo usa una clase personalizada:

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name = first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

Los puntos también son utilizados para llamar a métodos sobre los objetos. Por ejemplo, cada cadena de caracteres de Python tiene el métodos `upper()` y `isdigit()`, y puedes llamar a estos 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': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

Nota 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 los métodos que no requieran argumentos. (Explicaremos esta filosofía luego 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('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas', 'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

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

> **Listas de 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 como esto: cuando un sistema de plantillas encuentra un punto en una variable, este intenta la siguiente búsqueda en este orden:

- Diccionario (por ej. `foo["bar"]`)
- Atributo (por ej. `foo.bar`)
- Llamada de método (por ej. `foo.bar()`)
- 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 `{{ person.name.upper }}`, el cual se traslada en una búsqueda de diccionario (`person['name']`) y luego en una llamada a un método (`upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SALLY is 43 years old.'
```

Comportamiento de la llamada a los métodos

La llamada a los métodos es ligeramente más compleja que los otros tipos de búsquedas. 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 es `True`. Si la excepción *tiene* el atributo `silent_variable_failure`, la variable será renderizada como un string vacío, por ejemplo:

  ```
  >>> t = Template("My name is {{ person.first_name }}.")
  >>> class PersonClass3:
  ...     def first_name(self):
  ...         raise AssertionError, "foo"
  >>> p = PersonClass3()
  >>> t.render(Context({"person": p}))
  Traceback (most recent call last):
  ...
  AssertionError: foo

  >>> class SilentAssertionError(AssertionError):
  ...     silent_variable_failure = True
  >>> class PersonClass4:
  ...     def first_name(self):
  ...         raise SilentAssertionError
  >>> p = PersonClass4()
  >>> t.render(Context({"person": p}))
  "My name is ."
  ```

- La llamada a un método funcionará sólo si el método no requiere argumentos. En otro caso, el sistema se moverá a la próxima 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, tienes objeto `BankAccount` que tiene un método `delete()`. Una plantilla no debería permitir incluir algo como `{{ account.delete }}`.

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

```
def delete(self):
    # Delete the account
delete.alters_data = True
```

El sistema de plantillas no debería ejecutar cualquier método marcado de este modo. En otras palabras, si una plantilla incluye `{{ account.delete }}`, esta etiqueta no ejecutará el método `delete()`. Este fallará silenciosamente.

¿Cómo se manejan las variables inválidas?

Por omisión, si una variable no existe, el sistema de plantillas renderiza este como un string vacío, fallando silenciosamente, por ejemplo:

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
'Your name is .'
>>> t.render(Context({'var': 'hello'}))
'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
'Your name is .'
```

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

Ten en cuenta que es posible cambiar el comportamiento por omisión de Django en este sentido, ajustando la configuración de Django. Discutiremos esto más adelante en el **'Capítulo 10'** _.

4.2.5. Jugando con objetos Context

La mayoría de la veces, instancias un objeto `Context` pasando un diccionario completamente poblado a `Context`. Pero puedes agregar y quitar elementos desde un objeto `Context` una vez que este es instanciado, también, usando la sintaxis de los diccionarios estándares de Python:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

4.3. Etiquetas de plantillas básicas y filtros

Como hemos mencionamos, 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.

4.3.1. 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 sistemas mostrará todo entre `{% if %}` y `{% endif %}`, por ejemplo:

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% endif %}
```

La etiqueta `{% else %}` es opcional:

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% else %}
    <p>Get back to work.</p>
{% endif %}
```

> **Las "verdades" de Python**
>
> En Python, la lista vacía (`[]`), tuplas (`()`), diccionario (`{}`), string (`''`), cero (`0`), y el objeto especia `None` son `False` en un contexto booleano. Todo lo demás es `True`.

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

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches. (OK, so
    writing English translations of Boolean logic sounds
    stupid; it's not our fault.)
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% 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 athlete_list and coach_list or cheerleader_list %}
```

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

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

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

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

Ahí no hay una etiqueta `{% elif %}`. Usa etiquetas `{% if %}` anidadas para conseguir alguna cosa:

```
{% if athlete_list %}
    <p>Here are the athletes: {{ athlete_list }}.</p>
{% else %}
    <p>No athletes are available.</p>
```

```
    {% if coach_list %}
        <p>Here are the coaches: {{ coach_list }}.</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 `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

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

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

Es posible anidar etiquetas `{% for %}`:

```
{% for country in countries %}
    <h1>{{ country.name }}</h1>
    <ul>
    {% for city in country.city_list %}
        <li>{{ city }}</li>
    {% endfor %}
    </ul>
{% 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. (Ve a la sección "Filosofía y limitaciones" luego en este capítulo para comprender el razonamiento detrás de este decisión de diseño.)

La etiqueta `{% for %}` asigna la variable `forloop` mágica a la plantilla con el bucle. 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 item in todo_list %}
      <p>{{ forloop.counter }}: {{ item }}</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 asignada 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 object in objects%}
    {% if forloop.first%}<li class="first">{% else%}<li>{% endif%}
    {{ object }}
    </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 link in links%}{{ link }}{% if not forloop.last%} | {% endif%}{% endfor%}
```

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

```
Link1 | Link2 | Link3 | Link4
```

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

```
{% for country in countries%}
    <table>
    {% for city in country.city_list%}
        <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
        </tr>
    {% endfor%}
    </table>
{% endfor%}
```

La variable mágica `forloop` está sólo disponible dentro de bucles. 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 de tal manera de evitar sobreescribir la variable mágica `forloop`. Django expone ese contexto movido en `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 `user` y `currentuser` de la plantilla:

```
{% ifequal user currentuser%}
    <h1>Welcome!</h1>
{% endifequal%}
```

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

```
{% ifequal section 'sitenews'%}
    <h1>Site News</h1>
{% endifequal%}

{% ifequal section "community"%}
    <h1>Community</h1>
{% endifequal%}
```

Como `{% if %}`, la etiqueta `{% ifequal %}` admite un opcional `{% else %}`:

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

Sólo las variables de la plantilla, string, enteros y 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 comparados en `{% 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 vez de `{% ifequal %}`.

Comentarios

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

```
{# This is a comment #}
```

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):

```
This is a {# this is not
a comment #}
test.
```

4.3.2. 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:

```
{{ name|lower }}
```

Esto muestra el valor de `{{ name }}` después de aplicarle el filtro `lower`, el cual convierte el texto a minúscula. Usa un pipe (`|`) para aplicar el filtro.

Los filtros pueden estar en *cadena* -- eso es, la salida del uno de los filtros puede ser aplicada al próximo. Aquí un modismo común para escapar contenido del texto, y entonces convertir los saltos de líneas en etiquetas `<p>`:

```
{{ my_text|escape|linebreaks }}
```

Algunos filtros toman argumentos. Un filtro con argumento 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 F cubre el resto.

- `addslashes`: Agrega una con 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:

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

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

- `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__()`).

4.4. 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 con algunas de esas filosofías:

- *La lógica de negocios debe ser separada de la presentación lógica.* 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 sistemas 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 [3] 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 al 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 10'** _).
- *El objetivo no es 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 permiten.

4.5. Uso 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 `current_datetime` en `mysite.views`, la que comenzamos en el capítulo anterior. Se veía como esto:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now%s.</body></html>"% now
    return HttpResponse(html)
```

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

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

def current_datetime(request):
    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

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

Puedes primer considerar 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/mytemplate.html`:

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

def current_datetime(request):
    now = datetime.datetime.now()
    # Simple way of using templates from the filesystem.
    # This doesn't account for missing files!
```

```
fp = open('/home/djangouser/templates/mytemplate.html')
t = Template(fp.read())
fp.close()
html = t.render(Context({'current_date': now}))
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 descriptos en las secciones que siguen.

4.6. Cargadores de plantillas

Django provee una práctica y poderosa API [4] 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*.

El archivo de configuración de Django es el lugar para poner configuraciones para tu instancia de Django (aka [5] tu proyecto de Django). Es un simple módulo de Python con variables, una por cada configuración.

Cuando ejecutaste `django-admin.py startproject mysite` en el Capítulo 2, el script creó un archivo de configuración por omisión por ti, bien llamado `settings.py`. Échale un vistazo al contenido del archivo. Este contiene variables que se parecen a estas (no necesariamente en este orden):

```
DEBUG = True
TIME_ZONE = 'America/Chicago'
USE_I18N = True
ROOT_URLCONF = 'mysite.urls'
```

Éstas se explican por sí solas; las configuraciones y sus respectivos valores son simples variables de Python. Como el archivo de configuración es sólo un módulo plano de Python, puedes hacer cosas dinámicas como verificar el valor de una variable antes de configurar otra. (Esto también significa que debes evitar errores de sintaxis de Python en los archivos de configuración).

Cubriremos el archivo de configuración en profundidad en el Apéndice E, pero por ahora, veamos la variable de configuración `TEMPLATE_DIRS`. Esta variable le indica 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/mysite/templates',
)
```

Hay algunas cosas para notar:

- Puedes especificar cualquier directorio que quieras, siempre y cuando la cuenta de usuario en el cual se ejecuta el servidor web tengan acceso al directorio y 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 `mysite` que creaste en el Capítulo 2, si vienes siguiendo los ejemplos a lo largo del libro).
- ¡No olvides la coma al final del string del directorio de plantillas! Python requiere una coma en las tuplas de un solo elemento para diferenciarlas de una expresión de paréntesis. Esto es común en los usuarios nuevos.

  Si quieres evitar este error, puedes hacer `TEMPLATE_DIRS` una lista, en vez de una tupla, porque un solo elemento en una lista no requiere estar seguido de una coma:

```
TEMPLATE_DIRS = [
    '/home/django/mysite/templates'
]
```

Una tupla es un poco más correcta semánticamente que una lista (las tuplas no pueden cambiar luego de ser creadas, y nada podría cambiar las configuraciones una vez que fueron leídas), nosotros recomendamos usar tuplas para la variable `TEMPLATE_DIRS`.

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

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

- Es más sencillo usar rutas absolutas (esto es, las rutas de directorios comienzan desde la raíz del sistema de archivos). Si quieres sen un poco más flexible e independiente, también, puedes tomar el hecho de que el archivo de configuración de Django es sólo código de 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 de Python "mágica" `__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 que usa la funcionalidad de carga de plantillas de Django, para no incluir la ruta a la plantilla. Volvamos a nuestra vista `current_datetime`, vamos a cambiar esta como sigue:

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

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    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 de dónde está la plantilla en el sistema de archivos, lo abre, y retorna un objeto `Template` compilado.

Si `get_template()` no puede encontrar la plantilla con el nombre pasado, esta levanta una excepción `TemplateDoesNotExist`. Para ver que cómo se ve eso, ejecutar el servidor de desarrollo de Django otra vez, como en el Capítulo 3, ejecutando `python manage.py runserver` en el directorio de tu proyecto de Django. Luego, escribe en tu navegador la página que activa la vista `current_datetime` (o sea, `http://127.0.0.1:8000/time/`). Asumiendo que tu variable de configuración `DEBUG` está asignada a `True` y todavía no has creado la plantilla `current_datetime.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.

Como probablemente puedas distinguir de los mensajes de error de la Figura 4-1, 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/current_date` 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 `current_datetime.html` en tu directorio de plantillas usando el siguiente código:

```
<html><body>It is now {{ current_date }}.</body></html>
```

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

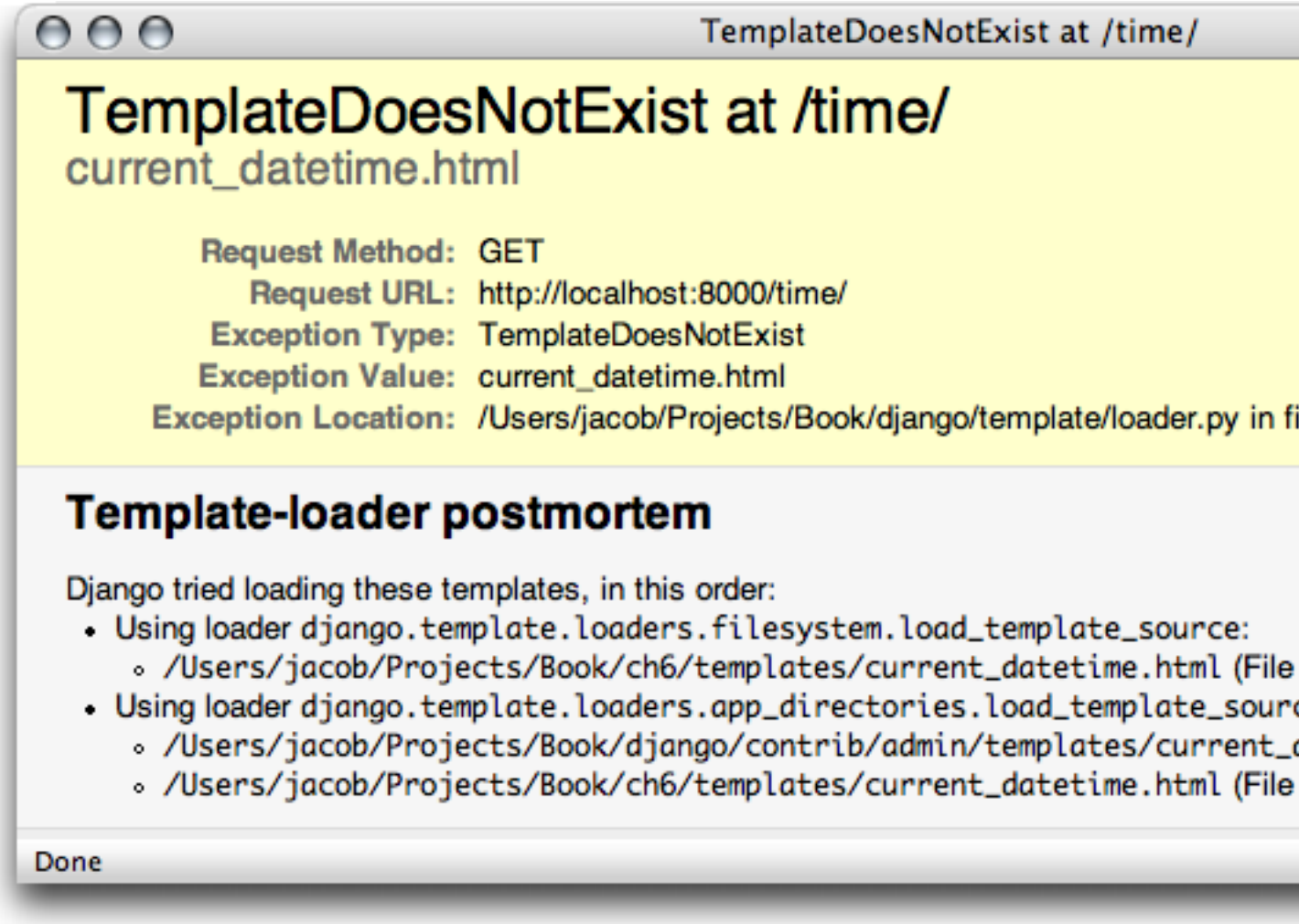

Figura 4.1: La página de error que se muestra cuando una plantilla no se encuentra

4.6.1. render_to_response()

Debido a que es común cargar una plantilla, rellenar un `Context`, y retornar un objeto `HttpResponse` con el resultado de la plantilla renderizada, Django provee un atajo que te deja hacer estas cosas en una línea de código. Este atajo es la función llamada `render_to_response()`, la cual se encuentra en el módulo `django.shortcuts`. La mayoría de las veces, usarás `render_to_response()` en vez de cargar las plantillas y crear los objetos `Context` y `HttpResponse` manualmente.

Aquí está el ejemplo actual `current_datetime` reescrito utilizando `render_to_response()`:

```
from django.shortcuts import render_to_response
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

¡Qué diferencia! Vamos paso a paso a través de los cambios del código:

- No tenemos que importar `get_template`, `Template`, `Context`, o `HttpResponse`. En vez de esto, importamos `django.shortcuts.render_to_response`. `import datetime` se mantiene.
- En la función `current_datetime`, seguimos calculando `now`, pero la carga de la plantilla, creación del contexto, renderización de esta, y de la creación de `HttpResponse` se encarga la llamada a `render_to_response()`. Como `render_to_response()` retorna un objeto `HttpResponse`, podemos simplemente `retornar` ese valor en la vista.

El primer argumento de `render_to_response()` 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_to_response()` utilizará un diccionario vacío.

4.6.2. El truco locals()

Considera nuestra última versión de `current_datetime`:

```
def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

Muchas veces, como en este ejemplo, buscarás tú mismo calcular algunos valores, guardando ellos en variables (por ej. `now` en el código anterior), y pasando estas a la plantilla. Particularmente los programadores perezosos notarán que es ligeramente redundante tener esos nombres en variables temporales *y* tener nombres para las variables de la plantilla. No sólo que esto es redundante, sino que también hay que teclear más.

Entonces si eres uno de esos programadores perezosos y quieres ahorrar código particularmente conciso, puedes tomar la ventaja de la función built-in de Python llamada `locals()`. Esta retorna un diccionario mapeando todos los nombres de variables locales con sus valores. De esta manera, la vista anterior podría reescribirse como sigue:

```
def current_datetime(request):
    current_date = datetime.datetime.now()
    return render_to_response('current_datetime.html', locals())
```

Aquí, en vez de especificar manualmente el diccionario al contexto como antes, pasamos el valor de `locals()`, el cual incluye todas las variables definidas hasta ese punto en la ejecución de la función. Como una consecuencia, renombramos el nombre de la variable `now` a `current_date`, porque esta es la variable que especificamos en la plantilla. En este ejemplo, `locals()` no ofrece una *gran* mejora, pero esta técnica puede salvar un poco de tipeo si tienes plantillas con varias variables definidas -- o si eres perezoso.

Una cosa en la que tiene que tener cuidado cuando usas `locals()` es que esta incluye *todas* las variables locales, con lo cual quizás conste de más variables de las cuales quieres tener acceso en la plantilla. En el ejemplo anterior, `locals()` también incluirá `request`. Depende de tu aplicación saber si esto es de importancia.

La última cosa a considerar es que `locals()` provoca un poco sobrecarga, porque cuando es llamado, Python crea el diccionario dinámicamente. Si especificas el diccionario al contexto manualmente, evitas esta sobrecarga.

4.6.3. 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 9) 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('dateapp/current_datetime.html')
```

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

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

> **Nota**
>
> Los usuario de Windows, asegúrense de usar barras comunes en vez de barras invertidas. `get_template()` asume el estilo de designación de archivos de Unix.

4.6.4. La etiqueta de plantilla `include`

Ahora que vimos el mecanismo para cargar plantillas, podemos introducir una plantilla built-in 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 un `{% include %}` para eliminar lo duplicado.

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.

Si una plantilla no encuentra el nombre tomado, Django hará una de estas dos cosas:

- Si `DEBUG` es `True`, verás la excepción `TemplateDoesNotExist` sobre la página de error de Django.
- Si `DEBUG` es `False`, la etiqueta fallará silenciosamente, sin mostrar nada en el lugar de la etiqueta.

4.7. 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 del desarrollo web: ¿Cómo reducimos la duplicación y 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 descripta. 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 los hijos puedan sobreescribir.

Veamos un ejemplo de esto creando una plantilla completa para nuestra vista `current_datetime`, editando el archivo `current_datetime.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Esto se ve bien, pero ¿Qué sucede cuando queremos crear una plantilla para otra vista --digamos, ¿La vista `hours_ahead` del Capítulo 3? Si queremos hacer nuevamente una agradable, válida, y completa plantilla HTML, crearíamos algo como:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Claramente, estaríamos duplicando una cantidad de código HTML. Imagina si tendríamos más sitios típicos, incluyendo barra 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 usando includes en el servidor es sacar factor común de ambas 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 `header.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 `footer.html`:

```
    <hr>
    <p>Thanks for visiting my site.</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>My helpful timestamp site</h1>`-- pero ese título no puede encajar dentro de `header.html` porque `<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*, defines los pedazos que son *diferentes*.

El primer paso es definir una *plantilla base*-- un "esquelete" de tu página que las *plantillas hijas* llenaran luego. Aquí hay una platilla para nuestro ejemplo actual:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
```

```
<html lang="en">
<head>
    <title>{% block title%}{% endblock%}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content%}{% endblock%}
    {% block footer%}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock%}
</body>
</html>
```

Esta plantilla, que llamamos `base.html`, define un documento esqueleto HTML simple que usaremos para todas las páginas del sitio. Es trabajo de las plantillas hijas sobreescribir, agregar, dejar vacío el contenido de los bloques. (Si estás lo siguiendo desde casa, guarda este archivo en tu directorio de plantillas).

Usamos una etiqueta de plantilla aquí que no hemos visto antes: la etiqueta `{% block%}`. Todas las etiquetas `{% block%}` le indican al motor de plantillas que una plantilla hijo quizás sobreescriba esa porción de la plantilla.

Ahora que tenemos una plantilla base, podemos modificar nuestra plantilla existente `current_datetime.html` para usar esto:

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

{% block title%}The current time{% endblock%}

{% block content%}
<p>It is now {{ current_date }}.</p>
{% endblock%}
```

Como estamos en este tema, vamos a crear una plantilla para la vista `hours_ahead` del Capítulo 3. (Si lo estás siguiendo junto con el código, te dejamos cambiar `hours_ahead` para usar el sistema de plantilla). Así sería el resultado:

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

{% block title%}Future time{% endblock%}

{% block content%}
<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
{% endblock%}
```

¿No es hermoso? Cada plantilla contiene sólo el código que es *único* para esa plantilla. No necesita redundancia. Si necesitas hacer un cambio grande en el diseño del sitio, sólo cambia `base.html`, y todas las otras plantillas reflejarán el efecto inmediatamente.

Veamos cómo trabaja. Cuando cargamos una plantilla `current_datetime.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 `{% block title%}` será usado, así como `{% block content%}`.

Nota que desde la plantilla hija no definimos el bloque `footer`, entonces el sistema de plantillas usa el valor desde la plantilla padre. El contenido de la etiqueta `{% block%}` en la plantilla padre es siempre usado como un plan alternativo.

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. Crear 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. Crear una plantilla `base_SECTION.html` para cada "sección" de tu sitio (por ej. `base_photos.html` y `base_forum.html`). Esas plantillas heredan de `base.html` e incluyen secciones específicas de estilo/diseño.

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

Esta aproximación maximiza la reutilización de código y hace fácil el agregado de elementos para compartir áreas, como puede ser un navegador de sección.

Aquí hay algunos consejos para el trabajo con herencia de plantillas:

- Si usas `{% extends %}` en la plantilla, esta debe ser la primer 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 cual de los bloques usar.
- 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, pero también puede ser una variable, si no sabes el nombre de la plantilla padre hasta la ejecución. Esto te permite hacer cosas divertidas, dinámicas.

4.8. ¿Qué sigue?

Los sitios web más 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 de los datos y la lógica (de la misma manera que las vistas y las etiquetas permiten una separación de la lógica y la vista).

El **‘próximo capítulo‘_** cubre las herramientas que Django brinda para interactuar con la base de datos.

Duplicate explicit target name: “próximo capítulo”.

[2] N. del T.: hard-coded
[3] N. del T.: WYSIWYG: What you see is what you get (Lo que ves es lo que obtienes)
[4] N. del T.: API: Application Program Interface (Interfaz de programación de aplicaciones)
[5] N. del T.: aka: Also Know As (También conocido como)

Capítulo 5

Interactuar con una base de datos: Modelos

En el Capítulo 3, recorrimos los conceptos fundamentales de la construcción dinámica de sitios web con Django: 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 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 esta, y los muestra con un formato agradable en una página web. O, del mismo modo, el sitio puede proporcionar funcionalidad que permita a los visitantes del sitio poblar la base de datos por su propia cuenta.

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

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 de bases de datos y SQL para usar la capa de base de datos de Django, es altamente recomendado. Una introducción a estos conceptos está más allá del alcance de este libro, pero continúa leyendo si eres nuevo en el tema. Probablemente seas capaz de seguir adelante y captar los conceptos básicos en base al contexto).

5.1. 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 la base de datos en una vista. Esto es simple: sólo usa una biblioteca de Python existente para ejecutar una consulta SQL y haz algo con los resultados.

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

```
from django.shortcuts import render_to_response
import MySQLdb

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

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

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

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 mysite.books.models import Book

def book_list(request):
    books = Book.objects.order_by('name')
    return render_to_response('book_list.html', {'books': books})
```

Explicaremos este código enseguida en este capítulo. Por ahora, tengamos sólo una idea de cómo es.

5.2. El patrón de diseño MTV

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

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

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

> **¿Por qué el acrónimo?**
>
> El objetivo de definir en forma explícita patrones como MVC es principalmente simplificar la comunicación entre los desarrolladores. En lugar de tener que decir a tus compañeros de trabajo, "Vamos a hacer una abstracción del acceso a la base de datos, luego vamos a tener 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 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 el modelos y las plantillas.

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

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

5.3. 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 iniciales: necesitamos indicarle a Django qué servidor de base de datos usar y cómo conectarse con el mismo.

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

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

```
DATABASE_ENGINE = ''
DATABASE_NAME = ''
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

Aquí hay un resumen de cada propiedad.

- `DATABASE_ENGINE` le indica a Django qué base de datos utilizar. Si usas una base de datos con Django, `DATABASE_ENGINE` debe configurarse con un string de los mostrados en la Tabla 5-1.

Cuadro 5.1: Configuración de motores de base de datos

Configuración	Base de datos	Adaptador requerido
`postgresql`	PostgreSQL	`psycopg` version 1.x, http://www.djangoproject.com/r/python-pgsql/1/.
`postgresql_psycopg2`	PostgreSQL	`psycopg` versión 2.x, http://www.djangoproject.com/r/python-pgsql/.
`mysql`	MySQL	`MySQLdb`, http://www.djangoproject.com/r/python-mysql/.
`sqlite3`	SQLite	No necesita adaptador si se usa Python 2.5+. En caso contrario, `pysqlite`, http://www.djangoproject.com/r/python-sqlite/.
`ado_mssql`	Microsoft SQL Server	`adodbapi` version 2.0.1+, http://www.djangoproject.com/r/python-ado/.
`oracle`	Oracle	`cx_Oracle`, http://www.djangoproject.com/r/python-oracle/.

Nota que cualquiera 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.

- `DATABASE_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/mydata.db'`).
- `DATABASE_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.
- `DATABASE_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.
- `DATABASE_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.
- `DATABASE_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.

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 `python manage.py shell` dentro del directorio del proyecto de Django y el más 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 hacer consultas a la base de datos: Django necesita saber cuales son los archivos de configuraciones a usar para obtener la información de la conexión a la base de datos.

Detrás de escena, `python 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 subtítulo lo cubriremos luego. Por ahora, usa `python manage.py shell` cuando necesites hacer modificaciones específicas a Django.

Una vez que hayas entrado al shell, escribe estos comando 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.

Cuadro 5.2: Mensajes de error de configuración de la base de datos

Mensaje de error	**Solución**
You haven't set the DATABASE_ENGINE setting yet.	Configura la variable `DATABASE_ENGINE` con otra cosa que un string vacío.
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`).
_____ isn't an available database backend.	Configura la variable `DATABASE_ENGINE` con un motor válido descrito previamente. ¿Habrás cometido un error de tipeo?
database _____ does not exist	Cambia la variable `DATABASE_NAME` para que *apunte* a una base de datos existente, o ejecuta la sentencia `CREATE DATABASE` apropiada para crearla.

Cuadro 5.2: Mensajes de error de configuración de la base de datos

Mensaje de error	Solución
role _____ does not exist	Cambia la variable `DATABASE_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 `DATABASE_HOST` y `DATABASE_PORT` estén configurados correctamente y que el servidor esté corriendo.

5.4. 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*? 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, 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 una funcionalidad de Django, típicamente incluye modelos y vistas, que conviven en un solo paquete de Python.

  Por ejemplo, Django incluye un número de aplicaciones, tales como un sistema de comentarios y una interfaz de administración automática. Una cosa clave para notar sobre estas aplicaciones es que son portables y reusables en múltiples proyectos.

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

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

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

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

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

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

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

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

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

5.5. 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 las escenas y retornar estructuras de datos convenientes en Python representando las filas de tus tablas de la base de datos. Django también usa modelos para representar conceptos de alto nivel que no necesariamente pueden ser manejados por SQL.

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

- La introspección requiere ***overhead*** y es imperfecta. Con el objetivo de proveer una API conveniente de acceso a los datos, Django necesita conocer *de alguna forma* la capa de la base de datos, y hay dos formas de lograr esto. La primera sería describir explícitamente los datos en Python, y la segunda sería la introspección de la base de datos en tiempo de ejecución para determinar el modelo de la base de datos.

  La segunda forma parece clara, porque los metadatos sobre tus tablas vive en un único lugar, pero introduce algunos problemas. Primero, introspeccionar una base de datos en tiempo de ejecución obviamente requiere overhead. Si el framework tuviera que introspeccionar la base de datos cada vez que se procese una petición, o incluso cuando el servidor web sea inicializado, esto podría provocar un nivel de overhead inaceptable. (Mientras algunos creen que el nivel de overhead es aceptable, los desarrolladores de Django apuntan a quitar del framework tanto overhead como sea posible, y esta aproximación hace que Django sea más rápido que los frameworks competidores de alto nivel en mediciones de desempeño). Segundo, algunas bases de datos, notablemente viejas versiones de MySQL, no guardan suficiente metadatos para asegurarse una completa introspección.
- Escribir Python es divertido, y dejar todo en Python limita el número de veces que tu cerebro tiene que realizar un "cambio de contexto". Si te mantienes en un solo entorno/mentalidad de programación tanto tiempo como sea posible, ayuda para la productividad. Teniendo que escribir SQL, luego Python, y luego SQL otra vez es perjudicial.
- Tener modelos de datos guardados como código en vez de en tu base de datos hace fácil dejar tus modelos bajo un control de versiones. De esta forma, puedes fácilmente dejar rastro de los cambios a tu capa de modelos.
- SQL permite sólo un cierto nivel de metadatos acerca de un ***layout*** de datos. La mayoría de sistemas de base de datos, por ejemplo, no provee un tipo de datos especializado para representar una dirección web o de email. Los modelos de Django sí. La ventaja de un tipo de datos de alto nivel es la alta productividad y la reusabilidad de código.
- SQL es inconsistente a través de distintas plataformas. Si estás redistribuyendo una aplicación web, por ejemplo, es mucho más pragmático distribuir un módulo de Python que describa tu capa de datos que separar conjuntos de sentencias `CREATE TABLE` para MySQL, PostgreSQL y SQLite.

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

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

5.6. Tu primer modelo

Como ejemplo continuo en este capítulo y el siguiente, nos enfocaremos en una configuración de datos básica sobre libro/autor/editor. Usamos esto como ejemplo porque las relaciones conceptuales entre libros, autores y editores son bien conocidas, y es una configuración de datos comúnmente utilizada en libros de texto introductorios de SQL. Por otra parte, ¡estás leyendo un libro que fue escrito por autores y producido por un editor!

Supondremos los siguientes conceptos, campos y relaciones:

- Un autor tiene un título (ej.: Sr. o Sra.), nombre, apellido, dirección de correo electrónico y una foto tipo carnet.
- Un editor tiene un nombre, una dirección, 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:

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

Examinemos rápidamente este código para conocer lo básico. La primer 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 `Publisher` es equivalente a la siguiente tabla (asumiendo la sintaxis de PostgreSQL para `CREATE TABLE`):

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" 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, `Book` tiene un `ManyToManyField` llamado `authors`. Esto significa que un libro tiene uno o más autores, pero la tabla de la base de datos `Book` no tiene una columna `authors`. En su lugar, Django crea una tabla adicional -- una "tabla de join" muchos-a-muchos -- que maneja la correlación entre libros y autores.

Para una lista completa de tipos de campo y opciones de sintaxis de modelos, ver 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 entera llamado `id`. Es un requerimiento el que cada modelo Django tenga una clave primaria de columna simple.

5.7. Instalando el modelo

Ya escribimos el código; creemos ahora las tablas en la base de datos. Para hacerlo, el primer paso es *activar* estos modelos en nuestro proyecto Django. Hacemos esto agregando la aplicación `books` a la lista de aplicaciones instaladas en el archivo de configuración.

Edita el archivo `settings.py` otra vez, y examina la variable de configuración `INSTALLED_APPS`. `INSTALLED_APPS` le indica a Django qué aplicaciones están activadas para un proyecto determinado. Por omisión, se ve como esto:

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

Temporalmente, comenta estos cuatro strings poniendo un carácter (#) al principio. (Están incluidos por omisión porque es frecuente usarlas, pero las activaremos y las discutiremos más adelante) Cuando termines, modifica las configuraciones por omisión de `MIDDLEWARE_CLASSES` y `TEMPLATE_CONTEXT_PROCESSORS`. Éstas dependen de algunas de las aplicaciones que hemos comentado. Entonces, agrega `'mysite.books'` a la lista `INSTALLED_APPS`, de manera que la configuración termine viéndose así:

```
MIDDLEWARE_CLASSES = (
#    'django.middleware.common.CommonMiddleware',
#    'django.contrib.sessions.middleware.SessionMiddleware',
#    'django.contrib.auth.middleware.AuthenticationMiddleware',
#    'django.middleware.doc.XViewMiddleware',
)

TEMPLATE_CONTEXT_PROCESSORS = ()
#...

INSTALLED_APPS = (
    #'django.contrib.auth',
    #'django.contrib.contenttypes',
    #'django.contrib.sessions',
    #'django.contrib.sites',
    'mysite.books',
)
```

(Como aquí estamos tratando con una tupla de un solo elemento, no olvides la coma 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)

`'mysite.books'` se refiere a la aplicación `books` 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.

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:

```
python manage.py validate
```

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 `python manage.py validate`. Tiende a capturar todos los problemas comunes del modelo.

Si tus modelos son válidos, ejecuta el siguiente comando para que Django genere sentencias `CREATE TABLE` para tus modelos en la aplicación `books` (con sintaxis resaltada en colores disponible si estás usando Unix):

```
python manage.py sqlall books
```

En este comando, `books` es el nombre de la aplicación. Es lo que hayas especificado cuando ejecutaste el comando `manage.py startapp`. Cuando ejecutes el comando, debes ver algo como esto:

```
BEGIN;
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL
);
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "salutation" varchar(10) NOT NULL,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL,
    "headshot" varchar(100) NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE INDEX books_book_publisher_id ON "books_book" ("publisher_id");
COMMIT;
```

Observa lo siguiente:

- Los nombres de tabla se generan automáticamente combinando el nombre de la aplicación (`books`) y el nombre en minúsculas del modelo (`publisher`, `book`, y `author`). Puedes sobreescribir 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 sobreescribir esto.
- Por convención, Django agrega `"_id"` al nombre de campo de las claves foráneas. Como ya puedes imaginar, también puedes sobreescribir 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 `sqlall` 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 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. Ejecuta el comando `syncdb` de esta manera:

```
python manage.py syncdb
```

Verás algo como esto:

```
Creating table books_publisher
Creating table books_book
Creating table books_author
Installing index for books.Book model
```

El comando `syncdb` es una simple sincronización de tus modelos hacia tu base de datos. El mismo 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. Observa que `syncdb` *no* sincroniza los cambios o eliminaciones de los modelos; si haces un cambio o modificas un modelo, y quieres actualizar la base de datos, `syncdb` no maneja esto. (Más sobre esto después.)

Si ejecutas `python manage.py syncdb` de nuevo, nada sucede, porque no has agregado ningún modelo a la aplicación `books` ni has incorporado ninguna aplicación en `INSTALLED_APPS`. Ergo, siempre es seguro ejecutar `python manage.py syncdb` -- no hará desaparecer cosas.

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 linea de comando ejecutar, dependiendo de tu configuración `DATABASE_SERVER`. Esto último es casi siempre más conveniente.

5.8. 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 `python manage.py shell` y escribiendo lo siguiente:

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Addison-Wesley', address='75 Arlington Street',
...     city='Boston', state_province='MA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

Estas pocas líneas logran bastantes resultados. Estos son los puntos salientes:

- Para crear un objeto, sólo importa la clase del modelo apropiada 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 `Publisher.objects`. Busca una lista de todos los objetos `Publisher` en la base de datos con la sentencia `Publisher.objects.all()`. Detrás de escenas, Django ejecuta aquí una sentencia SQL `SELECT`.

Naturalmente, puedes hacer mucho con la API de base de datos de Django -- pero primero, tengamos cuidado de una pequeña incomodidad.

5.9. Agregando strings de representación del modelo

Cuando imprimimos la lista de editores, todo lo que obtuvimos fue esta salida poco útil que hacía difícil distinguir los objetos `Publisher`:

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

Podemos arreglar esto fácilmente agregando un método llamado `__str__()` a nuestro objeto `Publisher`. Un método `__str__()` le dice a Python como mostrar la representación "string" de un objeto. Puedes ver esto en acción agregando un método `__str__()` a los tres modelos:

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

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

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

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

Como puedes ver, un método `__str__()` puede hacer lo que sea que necesite hacer para devolver una representación textual. Aquí, los métodos `__str__()` de `Publisher` y `Book` devuelven simplemente el nombre y título del objeto respectivamente, pero el `__str__()` del `Author` es un poco más complejo -- junta los campos `first_name` y `last_name`. El único requerimiento para `__str__()` es que devuelva un string. Si `__str__()` no devuelve un string -- si retorna, digamos, un entero -- entonces Python generará un `TypeError` con un mensaje como `"__str__ returned non-string"`.

Para que los cambios sean efectivos, sal del shell Python 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 `Publisher` es más fácil de entender:

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Addison-Wesley>, <Publisher: 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 lugares cuando necesita mostrar objetos.

Finalmente, observa que `__str__()` es un buen ejemplo de 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.

5.10. Insertando y actualizando 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, como:

```
>>> p = Publisher(name='Apress',
...         address='2855 Telegraph Ave.',
...         city='Berkeley',
...         state_province='CA',
...         country='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 book_publisher
    (name, address, city, state_province, country, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
     'U.S.A.', 'http://www.apress.com/');
```

Como el modelo `Publisher` 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
52    # esto será 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.name = 'Apress Publishing'
>>> p.save()
```

La sentencia `save()` del párrafo anterior resulta aproximadamente en la sentencia SQL siguiente:

```
UPDATE book_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

5.11. Seleccionar objetos

La creación y actualización de datos seguro es divertida, pero también es inútil sin una forma de tamizar los datos. Ya hemos visto una forma de examinar todos los datos de un determinado modelo:

```
>>> Publisher.objects.all()
[<Publisher: Addison-Wesley>, <Publisher: O'Reilly>, <Publisher: Apress Publishing>]
```

Eso se traslada a esto en SQL:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher;
```

Nota

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 más sobre el Zen de Python, intenta escribiendo `import this` en el prompt de Python.

Echemos un vistazo a cada parte de esta linea `Publisher.objects.all()`:

- En primer lugar, tenemos nuestro modelo definido, `Publisher`. 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 Apéndice B. Por ahora, todo lo que necesitas saber es que los administradores se encargan de 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 actualmente 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.

5.11.1. 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 con un subconjunto de los datos. Para ello usaremos el método `filter()`:

```
>>> Publisher.objects.filter(name="Apress Publishing")
[<Publisher: Apress Publishing>]
```

`filter()` toma argumentos de palabra clave que son traducidos en las cláusulas SQL `WHERE` apropiadas. El ejemplo anterior sería traducido en algo como:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name = 'Apress Publishing';
```

Puedes pasarle a `filter()` múltiples argumentos para reducir las cosas aún más:

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress Publishing>]
```

Esos 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, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A.' AND state_province = 'CA';
```

Notar que por omisión la búsqueda usa el operador SQL = para realizar búsquedas exactas. Existen también otros tipos de búsquedas:

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress Publishing>]
```

Notar el doble guión bajo entre `name` 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, name, address, city, state_province, country, website
FROM book_publisher
WHERE name LIKE '%press%';
```

Hay disponibles varios otos tipos de búsqueda, incluyendo `icontains` (`LIKE` no sensible a diferencias de mayúsculas/minúsculas), `startswith` y `endswith`, y `range` (consultas SQL `BETWEEN`). El Apéndice C describe en detalle todos esos tipos de búsqueda.

5.11.2. Obteniendo objetos individuales

En ocasiones desearás obtener un único objeto. Para esto existe el método `get()`:

```
>>> Publisher.objects.get(name="Apress Publishing")
<Publisher: 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 sean múltiples objetos causará una excepción:

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
AssertionError: get() returned more than one Publisher -- it returned 2!
```

Una consulta que no retorne objeto alguno también causará una excepción:

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
DoesNotExist: Publisher matching query does not exist.
```

5.11.3. Ordenando datos

A medida que juegas con los ejemplos anteriores, podrías descubrir que los objetos so 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 sus resultados, de manera que simplemente estamos recibiendo datos con algún orden arbitrario seleccionado por la base de datos.

Eso es, obviamente, un poco ***silly*** (tonto), no querríamos 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:

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

Esto no se ve muy diferente del ejemplo de `all()` anterior, pero el SQL incluye ahora un ordenamiento específico:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name;
```

Podemos ordenar por cualquier campo que deseemos:

```
>>> Publisher.objects.order_by("address")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]

>>> Publisher.objects.order_by("state_province")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

y por múltiples campos:

```
>>> Publisher.objects.order_by("state_provice", "address")
 [<Publisher: Apress Publishing>, <Publisher: O'Reilly>, <Publisher: Addison-Wesley>]
```

También podemos especificar un ordenamiento inverso antecediendo al nombre del campo un prefijo `-` (el símbolo menos):

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: 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 particular por el que usualmente desearás ordenar. Es esos casos Django te permite anexar al modelo un ordenamiento por omisión para el mismo:

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    class Meta:
        ordering = ["name"]
```

Este fragmento `ordering = ["name"]` le indica a Django que a menos que se proporcione un ordenamiento mediante `order_by()`, todos los editores deberán ser ordenados por su nombre.

> **¿Qué es este asunto de Meta?**
>
> 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. Examina el Apéndice B para conocer las opciones que puede poner bajo `Meta`.

5.11.4. Encadenando búsquedas

Has visto cómo puedes filtrar datos y has visto cómo ordenarlos. En ocasiones, por supuesto, vas a desear realizar ambas cosas. En esos casos simplemente "encadenas" las búsquedas entre sí:

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]
```

Como podrías esperar, esto se traduce a una consulta SQL conteniendo tanto un `WHERE` como un `ORDER BY`:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

Puedes encadenar consultas en forma consecutiva tantas veces como desees. No existe un límite para esto.

5.11.5. Rebanando 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:

```
>>> Publisher.objects.all()[0]
<Publisher: Addison-Wesley>
```

Esto se traduce, someramente, a:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name
LIMIT 1;
```

> **Y más...**
>
> Hemos sólo arañado la superficie del manejo de modelos, pero deberías ya conocer lo suficiente para entender todos los ejemplos del resto del libro. Cuando estés listo para aprender los detalles completos detrás de las búsquedas de objetos, échale una mirada al Apéndice C.

5.12. Eliminando objetos

Para eliminar objetos, simplemente llama al método `delete()` de tu objeto:

```
>>> p = Publisher.objects.get(name="Addison-Wesley")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>, <Publisher: O'Reilly>]
```

Puedes también borrar objetos al por mayor llamando a `delete()` en el resultado de algunas búsquedas:

```
>>> publishers = Publisher.objects.all()
>>> publishers.delete()
>>> Publisher.objects.all()
[]
```

> **Nota**
>
> Los borrados son *permanentes*, así que, ¡se cuidadoso!. En efecto, es usualmente una buena idea evitar eliminar objetos a menos que realmente tengas que hacerlo -- las base de datos relacionales no tiene una característica "deshacer" muy buena, y recuperar desde copias de respaldo es doloroso. A menudo es una buena idea agregar banderas "activo" a tus modelos de datos. Puedes buscar sólo objetos "activos", y simplemente fijar el campo activo a `False` en lugar de eliminar el objeto. Entonces, si descubres que has cometido un error, puedes simplemente volver a conmutar el estado de la bandera.

5.13. Realizando cambios en el esquema de una base de datos

Cuando presentamos el comando `syncdb` previamente en este capítulo, hicimos notar que `syncdb` simplemente crea tablas que todavía no existen en tu base de datos -- *no* sincroniza cambios en modelos ni borra modelos. Si agregas o modificas un campo de un modelo o si eliminas un modelo, será necesario que realices el cambio en tu base de datos en forma manual. Esta sección explica cómo hacerlo.

Cuando estás realizando cambios de esquema, es importante tener presente algunas características de la capa de base de datos de Django:

- Django se quejará estrepitosamente si un modelo contiene un campo que todavía no ha sido creado en la tabla de la base de datos. Esto causará un error la primera vez que uses la API de base de datos de Django para consultar la tabla en cuestión (esto es, ocurrirá en tiempo de ejecución y no en tiempo de compilación).
- A Django no le importa si una tabla de 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.

El realizar cambios al esquema de una base de datos es cuestión de cambiar las distintas piezas -- el código Python y la base de datos en sí misma -- en el orden correcto.

5.13.1. Agregando campos

Cuando se agrega un campo a una tabla/modelo en un entorno de producción, el truco es sacar ventaja del hecho que a Django no le importa si una tabla contiene columnas que no están definidas en el modelo. La estrategia es agregar la columna en la base de datos y luego actualizar el modelo Django para que incluya el nuevo campo.

Sin embargo, tenemos aquí un pequeño problema del huevo y la gallina, porque para poder saber cómo debe expresarse la nueva columna en SQL, necesitas ver la salida producida por el comando `manage.py sqlall` de Django, el cual requiere que el campo exista en el modelo. (Notar que *no* es un requerimiento el que crees tu columna con exactamente el mismo SQL que usaría Django, pero es una buena idea el hacerlo para estar seguros de que todo está en sincronía).

La solución al problema del huevo y la gallina es usar un entorno de desarrollo en lugar de realizar los cambios en un servidor de producción. (*Estás* usando un entorno de pruebas/desarrollo, ¿no es cierto?). Este es el detalle de los pasos a seguir.

Primero, realiza los siguientes pasos en el entorno de desarrollo (o sea, no en el servidor de producción):

1. Agrega el campo a tu modelo.
2. Ejecuta `manage.py sqlall [yourapp]` para ver la nueva sentencia `CREATE TABLE` para el modelo. Toma nota de la definición de la columna para el nuevo campo.
3. Arranca el shell interactivo de tu base de datos (por ej. `psql` o `mysql`, o puedes usar `manage.py dbshell`). Ejecuta una sentencia `ALTER TABLE` que agregue tu nueva columna.
4. (Opcional) Arranca el shell interactivo de Python con `manage.py shell` y verifica que el nuevo campo haya sido agregado correctamente importando el modelo y seleccionando desde la tabla (por ej. `MyModel.objects.all()[:5]`).

Entonces en el servidor de producción realiza los siguientes pasos:

1. Arranca el shell interactivo de tu base de datos.
2. Ejecuta la sentencia `ALTER TABLE` que usaste en el paso 3 de arriba.
3. Agrega el campo a tu modelo. Si estás usando un sistema de control de revisiones de código fuente y has realizado un *check in* de la modificación del paso 1 del trabajo en el entorno de desarrollo, entonces puedes actualizar el código (por ej. `svn update` si usas Subversion) en el servidor de producción.
4. Reinicia el servidor Web para que los cambios en el código surtan efecto.

Por ejemplo, hagamos un repaso de los que haríamos si agregáramos un campo `num_pages` al modelo `Book` descrito previamente en este capítulo. Primero, alteraríamos el modelo en nuestro entorno de desarrollo para que se viera así:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)

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

(Nota: Revisa el apartado "Agregando columnas NOT NULL" para conocer detalles importantes acerca de porqué hemos incluido `blank=True` y `null=True`).

Luego ejecutaríamos el comando `manage.py sqlall books` para ver la sentencia `CREATE TABLE`. La misma se vería similar a esto:

```
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL,
    "num_pages" integer NULL
);
```

La nueva columna está representada de la siguiente manera:

```
"num_pages" integer NULL
```

A continuación, arrancaríamos el shell interactivo de base de datos en nuestra base de datos de desarrollo escribiendo `psql` (para PostgreSQL), y ejecutaríamos la siguiente sentencia:

```
ALTER TABLE books_book ADD COLUMN num_pages integer;
```

Agregando columnas NOT NULL

Existe un detalle sutil aquí que merece ser mencionado. Cuando agregamos el campo `num_pages` a nuestro modelo, incluimos las opciones `blank=True` y `null=True`. Lo hicimos porque una columna de una base de datos contendrá inicialmente valores NULL desde el momento que la crees.

Sin embargo, es también posible agregar columnas que no puedan contener valores NULL. Para hacer esto, tienes que crear la columna como `NULL`, luego poblar los valores de la columna usando algunos valor(es) por omisión, y luego modificar la columna para activar el modificador `NOT NULL`. Por ejemplo:

```
BEGIN;
ALTER TABLE books_book ADD COLUMN num_pages integer;
UPDATE books_book SET num_pages=0;
ALTER TABLE books_book ALTER COLUMN num_pages SET NOT NULL;
COMMIT;
```

Si sigues este camino, recuerda que debes quitar `blank=True` y `null=True` de tu modelo.

Luego de la sentencia `ALTER TABLE`, verificaríamos que el cambio haya funcionado correctamente, para ello iniciaríamos el shell de Python y ejecutaríamos este código:

```
>>> from mysite.books.models import Book
>>> Book.objects.all()[:5]
```

Si dicho código no produjera errores, podríamos movernos a nuestro servidor de producción y ejecutaríamos la sentencia `ALTER TABLE` en la base de datos de producción. Entonces, actualizaríamos el modelo en el entorno de producción y reiniciaríamos el servidor Web.

5.13.2. Eliminando campos

Eliminar un campo de un modelo es mucho más fácil que agregar uno. Para borrar un campo, sólo sigue los siguientes pasos:

1. Elimina el campo de tu modelo y reinicia el servidor Web.
2. Elimina la columna de tu base de datos, usando un comando como este:

   ```
   ALTER TABLE books_book DROP COLUMN num_pages;
   ```

5.13.3. Eliminando campos Many-to-Many

Debido a que los campos many-to-many son diferentes a los campos normales, el proceso de borrado es diferente:

1. Elimina el campo `ManyToManyField` de tu modelo y reinicia el servidor Web.
2. Elimina la tabla many-to-many de tu base de datos, usando un comando como este:

   ```
   DROP TABLE books_books_publishers;
   ```

5.13.4. Eliminando modelos

Eliminar completamente un modelo es tan fácil como el eliminar un campo. Para borrar un modelo, sigue los siguientes pasos:

1. Elimina el modelo de tu archivo `models.py` y reinicia el servidor Web.
2. Elimina la tabla de tu base de datos, usando un comando como este:

   ```
   DROP TABLE books_book;
   ```

5.14. ¿Qué sigue?

Una vez que has definido tus modelos, el paso siguiente es el poblar tu base de datos con datos. Podrías tener datos legados, en cuyo caso el Capítulo 16 te aconsejará acerca de cómo integrar bases de datos heredadas. Podrías delegar en los 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. El **‘próximo capítulo‘_** está dedicado a la interfaz de administración de Django, la cual existe precisamente por esa razón.

Duplicate explicit target name: “próximo capítulo”.

Capítulo 6

El sitio de Administración Django

Para cierto tipo de Sitios Web, una *interfaz de administración* es una parte esencial de la infraestructura. Se trata de una interfaz basada en web, 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 los lectores, la herramienta que tus clientes utilizan para actualizar los comunicados de prensa en la web que construiste para ellos -- todos son ejemplos de interfaces de administración.

Aunque hay un problema con las interfaces de administración: es aburrido construirlas. El desarrollo web es divertido cuando estás desarrollando funcionalidades de lado público del sitio, pero construir interfaces de administración es siempre lo mismo. Tienes que autenticar usuarios, mostrar y manipular formularios, validar las entradas y demás. 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 los meta-datos en tus modelos para brindar una interfaz potente y lista para producción que los administradores del sitio podrán usar inmediatamente. Aquí discutimos cómo activar, usar y personalizar esta utilidad.

6.1. Activando la interfaz de administración

Pensamos que la interfaz de administración es la característica más atractiva de Django -- y la mayoría de Djangonautas están de acuerdo -- pero como no todo el mundo lo necesita, es una pieza opcional. Esto significa que hay que dar tres pasos para activar la interfaz de administración:

1. Agrega meta-datos de administración a tus modelos.

   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. Esto lo hacemos añadiendo al modelo una clase interna `Admin` (junto con la clase `Meta`, si es que hay una). Así que, para agregar una interfaz de administración a nuestro modelo `Book` del capítulo anterior, usamos:

   ```
   class Book(models.Model):
       title = models.CharField(maxlength=100)
       authors = models.ManyToManyField(Author)
       publisher = models.ForeignKey(Publisher)
       publication_date = models.DateField()
       num_pages = models.IntegerField(blank=True, null=True)

       def __str__(self):
           return self.title

       class Admin:
           pass
   ```

   La declaración de `Admin` marca la clase como poseedora de una interfaz de administración. Hay una serie de opciones que podemos incluir bajo `Admin`, pero por ahora vamos a limitarnos al comportamiento por defecto, así que escribimos `pass` para decirle a Python que la clase `Admin` está vacía.

Si estás siguiendo este ejemplo escribiendo tu propio código, probablemente sea buena idea agregar ahora declaraciones de `Admin` a las clases `Publisher` y `Author`.

2. Instalar la aplicación `admin`. Esto se hace agregando `"django.contrib.admin"` a tus `INSTALLED_APPS` de tu archivo de configuración `settings.py`.
3. Además, asegurate de que las aplicaciones `"django.contrib.sessions"`, `"django.contrib.auth"`, y `"django.contrib.contenttypes"` no están comentadas, ya que la aplicación `admin` depende de ellas. También descomenta todas las líneas de `MIDDLEWARE_CLASSES` configurando la tupla, y borra la definición de `TEMPLATE_CONTEXT_PROCESSOR` para permitir que tome los valores por defecto.
4. Ejecuta `python manage.py syncdb`. Este paso instalará las tablas de la base de datos que la interfaz de administración necesita.

   Nota

   Es probable que la primera vez que ejecutaste `syncdb` con `"django.contrib.auth"` en INSTALLED_APPS, te preguntara algo sobre crear un superusuario. Si no lo hiciste en ese momento, tendrás que ejecutar `django/contrib/auth/bin/create_superuser.py` para crear este usuario administrador. En caso contrario no serás capaz de identificarte para entrar a la interfaz de administración.

5. Agrega el patrón de URL en tu `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, los patrones de URL deberían terminar siendo algo así:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (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 0.97, using settings 'mysite.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 (`http://127.0.0.1:8000/admin/` en el ejemplo precedente), identificarte, y jugar un poco.

6.2. Usando 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 unas pocas notas sobre sus características.

Lo primero que verás es una página de identificación, como se muestra en la Figura 6-1.

Usarás el nombre de usuario y la clave que configuraste cuando agregaste tu superusuario. Una vez identificado, verás que puedes gestionar usuarios, grupos y permisos (veremos más sobre esto en breve).

Cada objeto al que se le dió una declaración `Admin` aparece en el índice de la página principal, como se muestra en la Figura 6-2

Los enlaces para agregar y modificar objetos llevan a dos páginas a las que nos referiremos como `listas de cambio` [6] y `formularios de edición` [7] de objetos:

Las listas de cambio son esencialmente páginas de índices de objetos en el sistema, como se muestra en la Figura 6-3.

Hay varias opciones que pueden controlar los campos que aparecen en esas listas y la aparición de características extra como campos de búsqueda e accesos directo a filtros predefinidos. Más adelante hablaremos sobre esto.

Los formularios de edición se usan para modificar objetos existente y crear nuevos (mira la Figura 6-4). Cada campo definido en tu modelo aparece aquí, y notarás que campos de tipos diferentes tienen diferentes controles. (Por ejemplo, los campos de fecha/hora tienen controles tipo calendario, las claves foráneas usan cajas de selección, etc.)

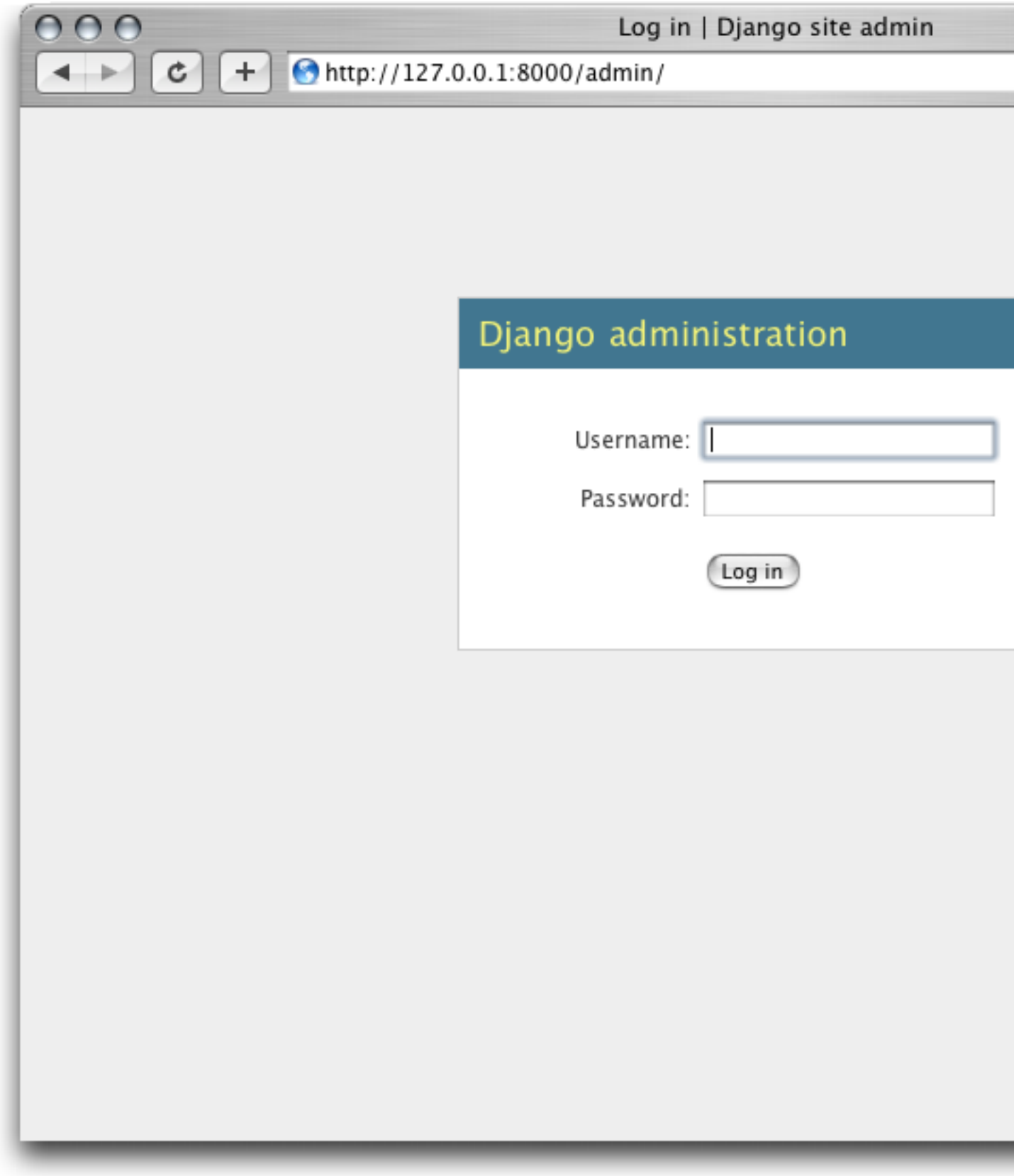

Figura 6.1: Pantalla de autenticación de Django.

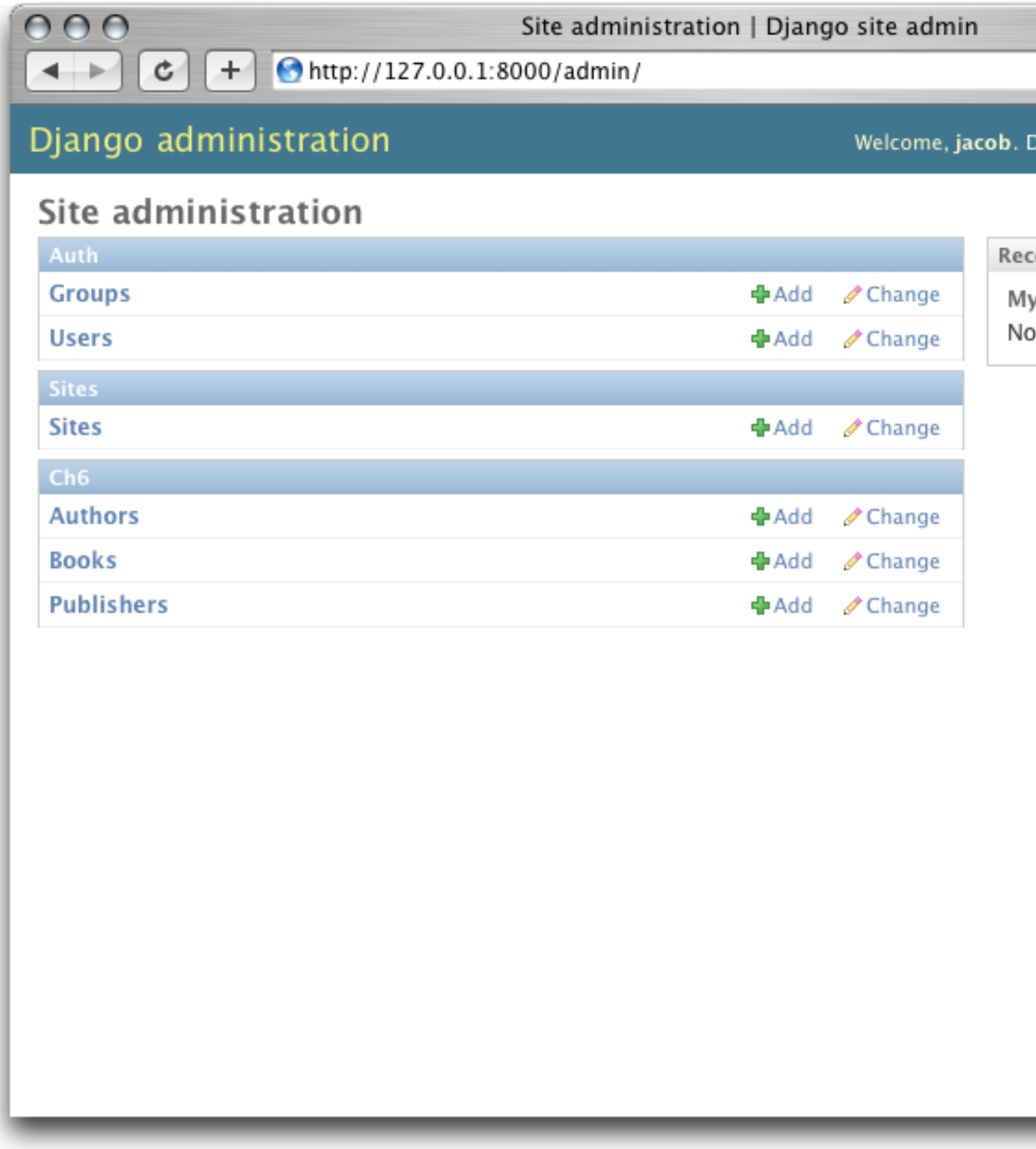

Figura 6.2: El índice principal de la Administración de Django.

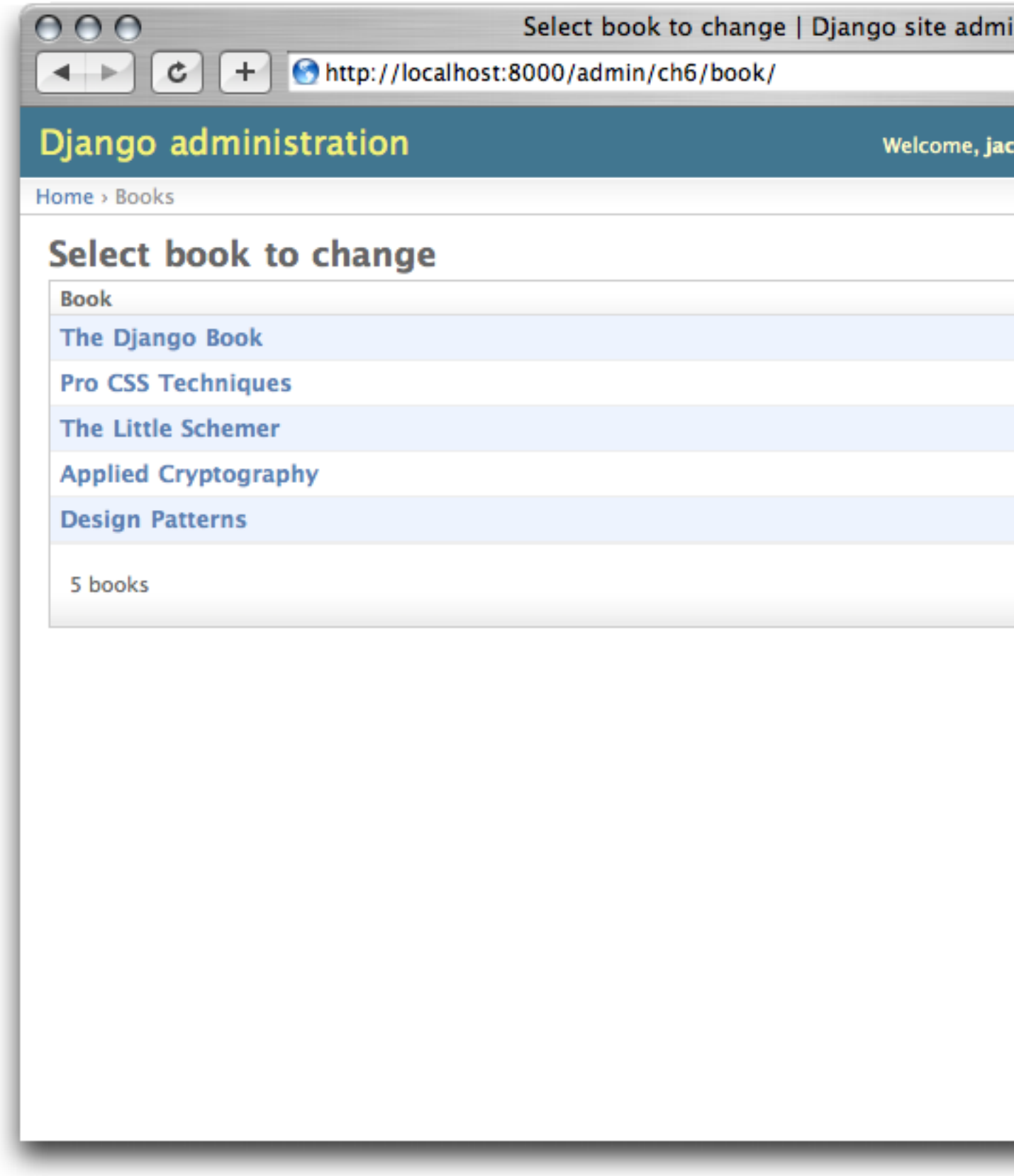

Figura 6.3: Una típica vista de lista de cambio

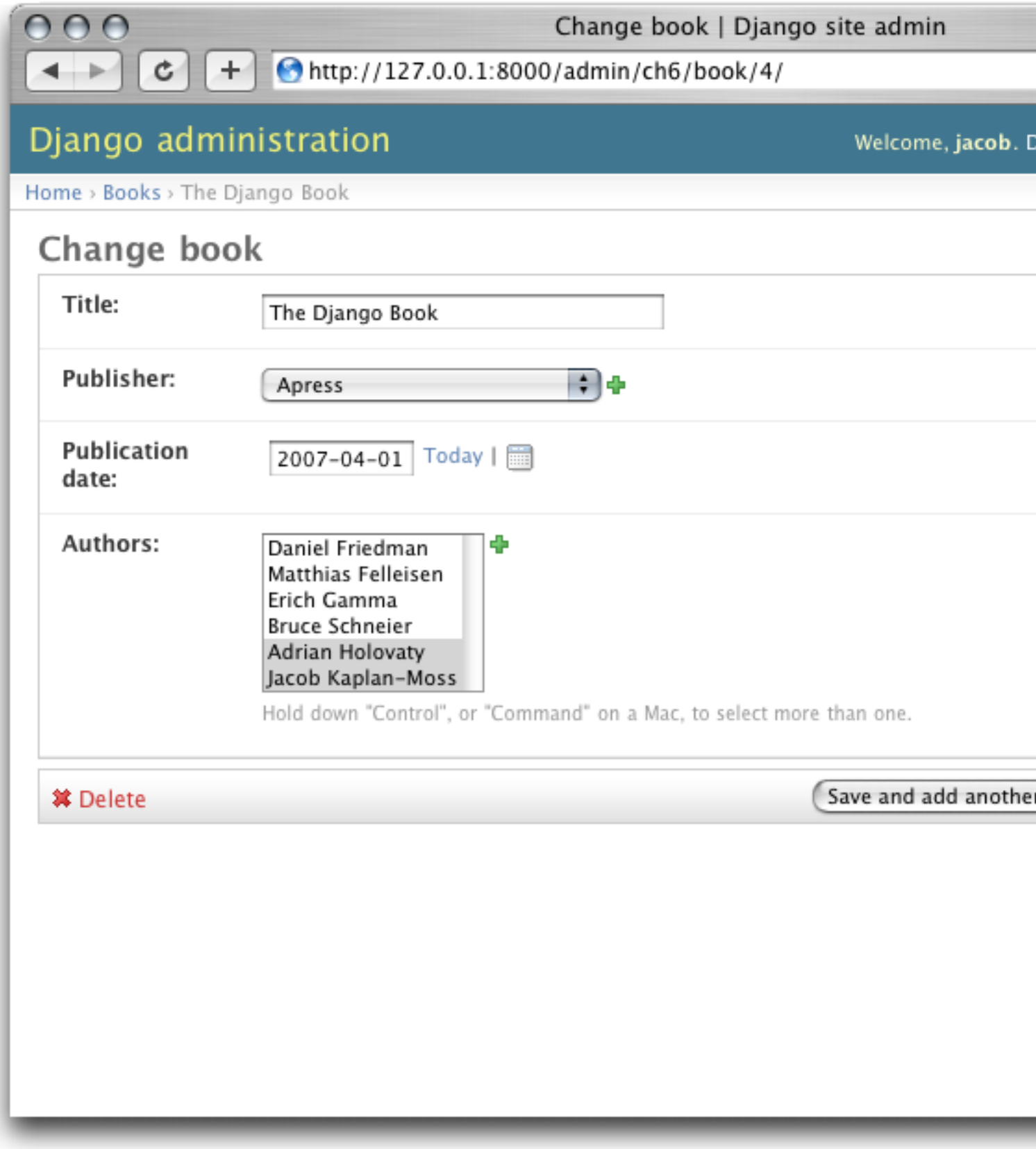

Figura 6.4: Un típico formulario de edición

Te darás cuenta que la interfaz de administración también controla por ti la validez de los datos ingresado. Intenta dejar un campo requerido en blanco o poner una fecha inválida en un campo de fecha, y verás esos avisos de esos errores cuando intentes guardar el objeto, como se muestra en la Figura 6-5.

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 click en este botón (mira la Figura 6-6).

Cuando eliminas un objeto existente, la interfaz de administración solicita una confirmación para prevenir costosos 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 (mira la Figura 6-7).

6.2.1. 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 darle 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 el los 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 "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 el Capítulo 12.), esta opción diferencia entre usuarios públicos y administradores.
- 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 superusuario" da al usuario completo e irrestricto acceso a todos los elementos de la interfaz de administración, y sus permisos regulares son ignorados.

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* [8], un permiso de *modificar* [9], y un permiso de *eliminar* [10]. 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 asignar 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.

6.3. Personalizando la interfaz de administración

Puedes personalizar el aspecto y la forma en que la interfaz de administración se comporta de varias maneras. En esta sección sólo vamos a cubrir algunas de ellas relacionadas con nuestro modelo `Libro`. El Capítulo 17 descubre la personalización de la interfaz de administración en detalle.

Como estamos ahora, la lista de cambio de nuestros libros sólo muestra la cadena de representación del modelo que agregamos con el método `__str__`

Esto funciona bien sólo para algunos libros, pero si tuviéramos cientos o miles de libros, se volvería tan difícil como encontrar una aguja en un pajar. Sin embargo, fácilmente podremos agregar algunas columnas, funciones de búsqueda y filtros y a esta interfaz. Cambia la declaración de `Admin` como sigue:

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

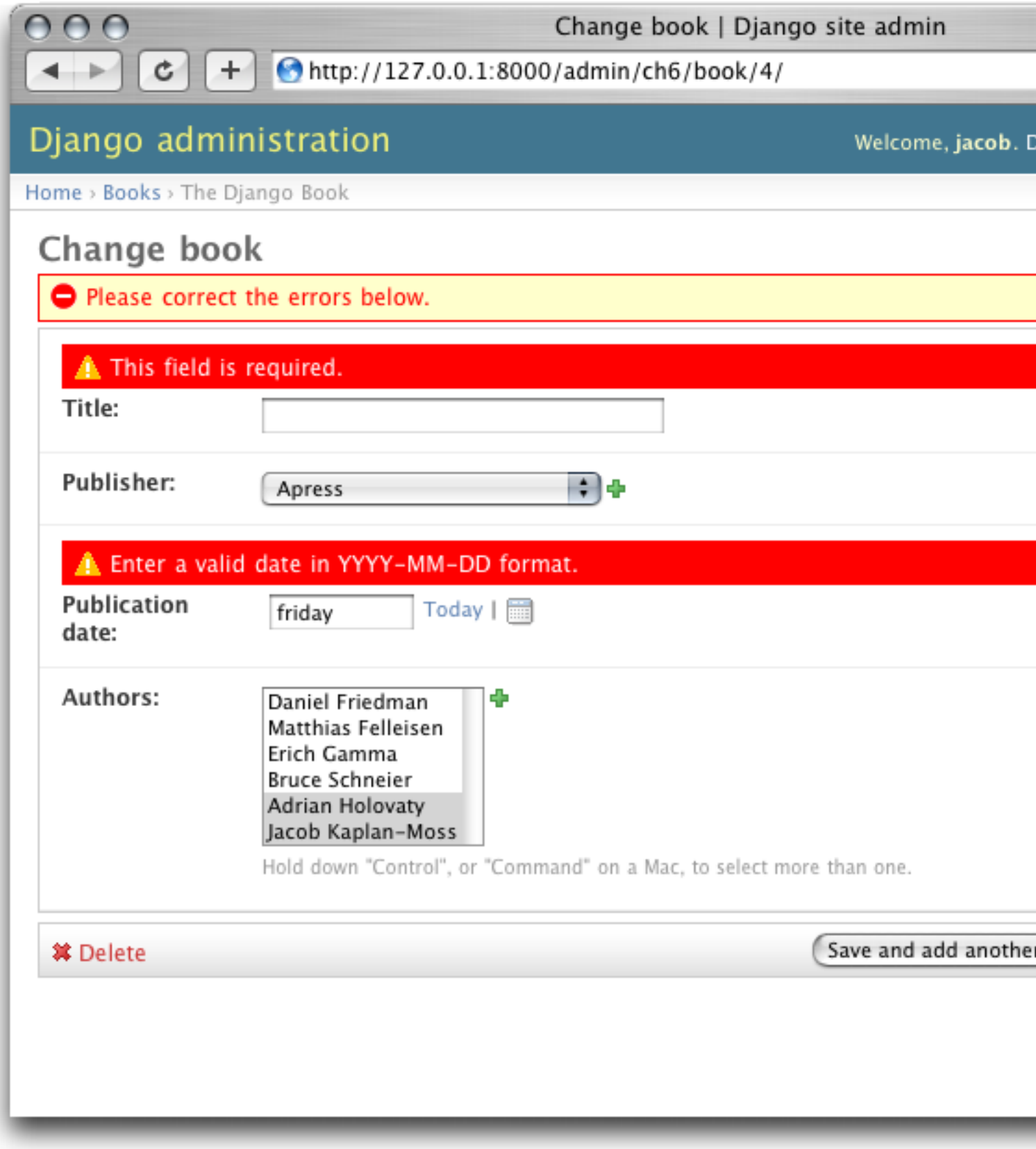

Figura 6.5: Un formulario de edición mostrando errores

Change history: The Django Book | Django site

http://127.0.0.1:8000/admin/ch6/book/4/history/

Django administration

Welcome, jac

Home › Books › The Django Book › History

Change history: The Django Book

Date/time	User	Action
Nov. 12, 2006, 11:21 a.m.	jacob	
Nov. 12, 2006, 11:22 a.m.	jacob	Changed publication date.
Nov. 12, 2006, 11:22 a.m.	jacob	Changed publication date.
Nov. 12, 2006, 11:22 a.m.	jacob	Changed publisher.
Nov. 12, 2006, 11:23 a.m.	jacob	Changed title and publisher.
Nov. 12, 2006, 11:23 a.m.	jacob	Changed title.

Figura 6.6: Página de historia de un objeto django.

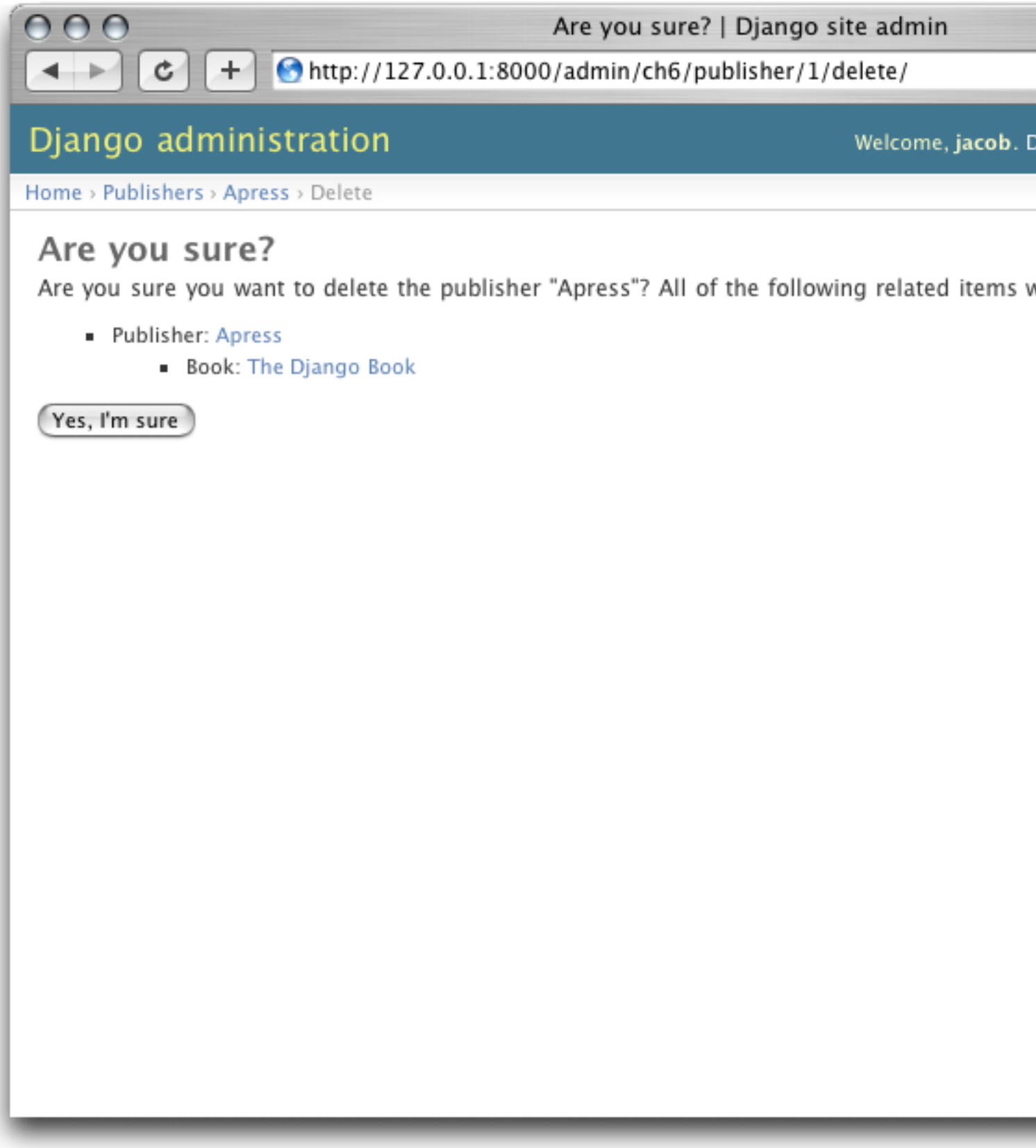

Figura 6.7: Una página de confirmación de eliminación de un objeto Django

```
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    class Admin:
        list_display = ('title', 'publisher', 'publication_date')
        list_filter = ('publisher', 'publication_date')
        ordering = ('-publication_date',)
        search_fields = ('title',)
```

Estas cuatro líneas de código cambian dramáticamente la interfaz de nuestra lista, como se muestra en la figura 6-8.

Cada una de estas líneas indica a la interfaz de administración que construya diferentes piezas de la interfaz:

- La opción `list_display` controla que columnas aparecen en la tabla de la lista. Por defecto, la lista de cambios muestra una sola columna que contiene la representación en cadena de caracteres del objeto. Aquí podemos cambiar eso para mostrar el título, el editor y la fecha de publicación.
- La opción `list_filter` crea una barra de filtrado del lado derecho de la lista. Estaremos habilitados a filtrar por fecha (que te permite ver sólo los libros publicados la última semana, mes, etc.) y por editor.

  Puedes indicarle a la interfaz de administración que filtre por cualquier campo, pero las claves foráneas, fechas, booleanos, y campos con un atributo de opciones `choices` son las que mejor funcionan. Los filtros aparecen cuando tienen al menos 2 valores de dónde elegir.
- La opción `ordering` controla el orden en el que los objetos son presentados en la interfaz de administración. Es simplemente una lista de campos con los cuales ordenar el resultado; anteponiendo un signo menos a un campo se obtiene el orden reverso. En este ejemplo, ordenamos por fecha de publicación con los más recientes al principio.
- Finalmente, la opción `search_fields` crea un campo que permite buscar texto. En nuestro caso, buscará el texto en el campo `título` (entonces podrías ingresar **Django** para mostrar todos los libros con "Django" en el título).

Usando estas opciones (y las otras descriptas en el capítulo 17) puedes, con sólo algunas líneas de código, hacer una interfaz de edición de datos realmente potente y lista para producción.

6.4. Personalizando 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 el Capítulo 4.)

Como explicamos en el Capítulo 4, 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 cualquiera del directorio de `TEMPLATE_DIRS` que estés usando. Por ejemplo, si tu `TEMPLATE_DIRS` incluye `"/home/misplantillas"`, entonces copia `django/contrib/admin/templates/admin/base_site` a `/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 el capítulo 10 para obtener más información sobre cómo funciona esto.

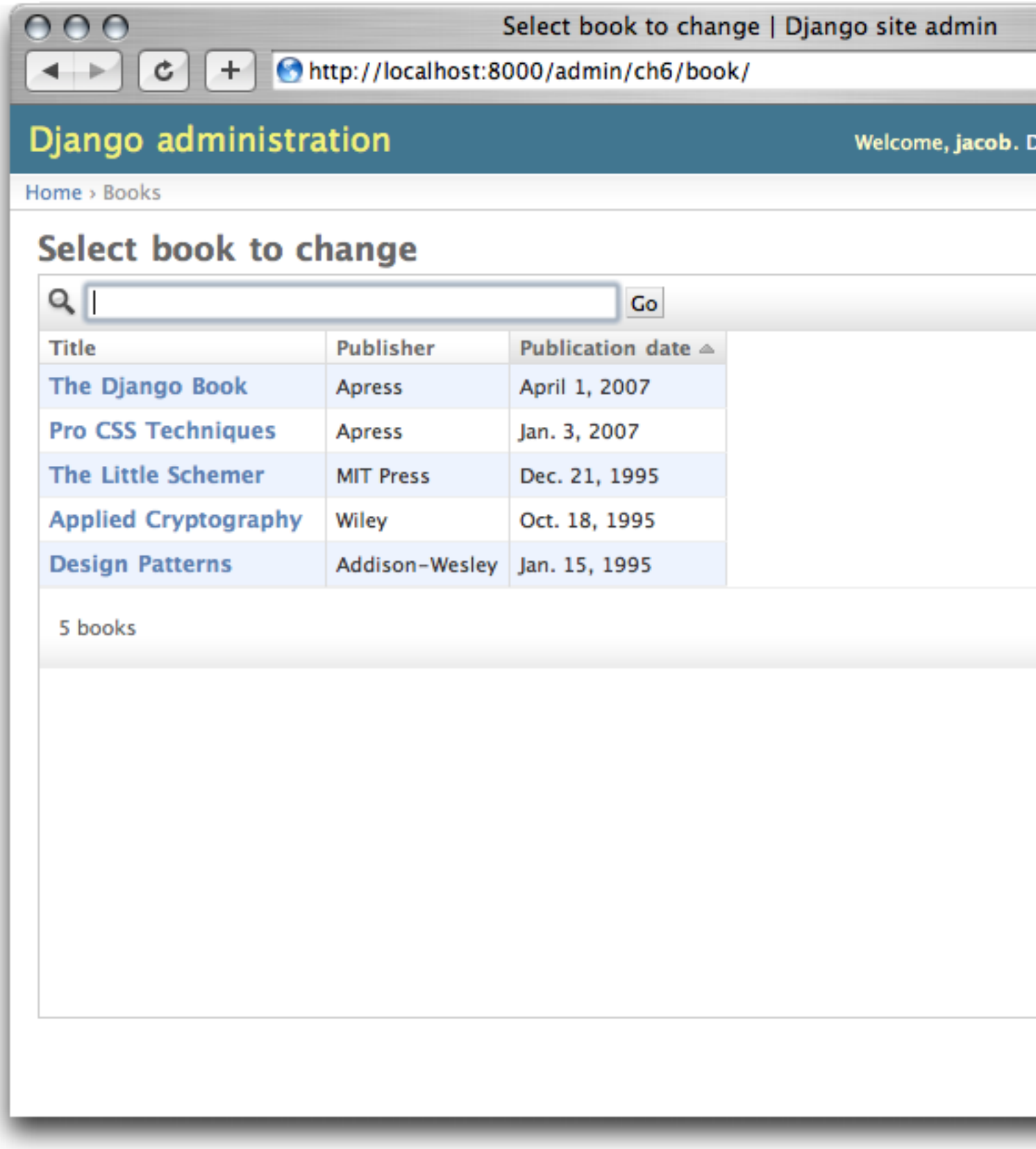

Figura 6.8: Página de lista de cambios modificada

6.5. Personalizando 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 código explícito en una plantilla no te satisface, puedes ver el **'Capítulo 10'_** para encontrar detalles sobre cómo implementar tu propias etiquetas de plantillas.

Django ofrece otro acceso directo en este apartado. Ejecuta el comando `python manage.py adminindex <aplicación>` para obtener un pedazo de código de plantilla para incluir en la página índice del administrador. Es un punto de partida bastante útil.

Para detalles completos sobre la personalización del sitio de administración de Django, mira el Capítulo 17.

6.6. 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 una característica típica online -- un reporte especial sobre la calidad del agua del acueducto municipal, pongamos -- 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 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.or` 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.

6.7. ¿Qué sigue?

Hasta ahora hemos creado algunos modelos y configurado una interfaz de primera clase para modificar datos. En el **'próximo capítulo'_**, nos meteremos en el verdadero guiso del desarrollo Web: creación y procesamiento de formularios.

Duplicate explicit target name: “próximo capítulo”.

[6] N. del T.: *change list* es el nombre que recibe en inglés
[7] N. del T.: *edit forms* es el nombre que recibe en inglés
[8] N. del T.: En el control de selección de permisos aparece como *Can add*
[9] N. del T.: *Can change*
[10] N. del T.: *Can delete*

Capítulo 7

Procesamiento de formularios

Autor invitado: Simon Willison

Si has estado siguiendo el capítulo anterior, ya deberías tener un sitio completamente funcional, aunque un poco simple. En este capítulo trataremos con la próxima pieza del juego: cómo construir vistas que obtienen entradas desde los usuarios.

Comenzaremos haciendo un simple formulario de búsqueda "a mano", viendo cómo manejar los datos suministrados al navegador. Y a partir de ahí, pasaremos al uso del *framework* de formularios que trae Django.

7.1. Búsquedas

En la Web todo se trata de búsquedas. Dos de los casos de éxito más grandes, Google y Yahoo, han construido sus empresas multimillonarias alrededor de las búsquedas. Casi todos los sitios observan un gran porcentaje de tráfico viniendo desde y hacia sus páginas de búsqueda. A menudo, la diferencia entre el éxito y el fracaso de un sitio, lo determina la calidad de su búsqueda. Así que sería mejor que agreguemos un poco de búsqueda a nuestro pequeño sitio de libros, ¿no?

Comenzaremos agregando la vista para la búsqueda a nuestro URLconf (`mysite.urls`). Recuerda que esto se hace agregando algo como `(r'^search/$', 'mysite.books.views.search')` al conjunto de URL patterns (patrones).

A continuación, escribiremos la vista `search` en nuestro módulo de vistas (`mysite.books.views`):

```
from django.db.models import Q
from django.shortcuts import render_to_response
from models import Book

def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })
```

Aquí han surgido algunas cosas que todavía no vimos. La primera, ese `request.GET`. Así es cómo accedes a los datos del GET desde Django; Los datos del POST se acceden de manera similar, a través de un objeto llamado `request.POST`. Estos objetos se comportan exactamente como los diccionarios estándar de Python, y tienen además otras capacidades, que se cubren en el apéndice H.

¿Qué son estos datos del GET y del POST?

GET y POST son los dos métodos que emplean los navegadores para enviar datos a un servidor. Los encontrarás con frecuencia en los elementos *form* de HTML:

```
<form action="/books/search/" method="get">
```

Esto le indica al navegador que suministre los datos del formulario a la URL `/books/search/` empleando el método GET.
Hay diferencias de semántica importantes entre el GET y el POST, que no vamos a ver ahora mismo, pero diríjete a http://www.w3.org/2001/tag/doc/whenToUseGet.html si quieres aprender más.

Así que la línea:

```
query = request.GET.get('q', '')
```

busca un parámetro del GET llamado `q` y retorna una cadena de texto vacía si este parámetro no fue suministrado. Observa que estamos usando el método `get()` de `request.GET`, algo potencialmente confuso. Este método `get()` es el mismo que posee cualquier diccionario de Python. Lo estamos usando aquí para ser precavidos: *no* es seguro asumir que `request.GET` tiene una clave `'q'`, así que usamos `get('q', '')` para proporcionar un valor por omisión, que es `''` (el string vacío). Si hubiéramos intentado acceder a la variable simplemente usando `request.GET['q']`, y `q` no hubiese estado disponible en los datos del GET, se habría lanzado un `KeyError`.

Segundo, ¿qué es ese `Q`? Los objetos `Q` se utilizan para ir construyendo consultas complejas -- en este caso, estamos buscando los libros que coincidan en el título o en el nombre con la consulta. Técnicamente, estos objetos `Q` consisten de un QuerySet, y puede leer más sobre esto en el apéndice C.

En estas consultas, `icontains` es una búsqueda 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.

Dado que estamos buscando en campos de muchos-a-muchos, es posible que un libro se obtenga más de una vez (por ej: un libro que tiene dos autores, y los nombres de ambos concuerdan con la consulta). Al agregar `.distinct()` en el filtrado, se eliminan los resultados duplicados.

Todavía no hay una plantilla para esta vista. Esto lo solucionará:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Search{% if query%} Results{% endif%}</title>
</head>
<body>
  <h1>Search</h1>
  <form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>

  {% if query%}
    <h2>Results for "{{ query|escape }}":</h2>

    {% if results%}
      <ul>
      {% for book in results%}
        <li>{{ book|escape }}</l1>
      {% endfor%}
      </ul>
    {% else%}
      <p>No books found</p>
    {% endif%}
  {% endif%}
</body>
</html>
```

A esta altura, lo que esto hace debería ser obvio. Sin embargo, hay unas pocas sutilezas que vale la pena resaltar:

- *action* es `.` en el formulario, esto significa "la URL actual". Esta es una buena práctica estándar: no utilices vistas distintas para la página que contiene el formulario y para la página con los resultados; usa una página única para las dos cosas.
- Volvemos a insertar el texto de la consulta en el `<input>`. Esto permite a los usuarios refinar fácilmente sus búsquedas sin tener que volver a teclear todo nuevamente.
- En todo lugar que aparece `query` y `book`, lo pasamos por el filtro `escape` para asegurarnos de que cualquier búsqueda potencialmente maliciosa sea descartada antes de que se inserte en la página

  ¡Es *vital* hacer esto con todo el contenido suministrado por el usuario! De otra forma el sitio se abre a ataques de cross-site scripting (XSS). El **'Capítulo 19'_** discute XSS y la seguridad con más detalle.
- En cambio, no necesitamos preocuparnos por el contenido malicioso en las búsquedas de la base de datos -- podemos pasar directamente la consulta a la base de datos. Esto es posible gracias a que la capa de base de datos de Django se encarga de manejar este aspecto de la seguridad por ti.

Ahora ya tenemos la búsqueda funcionando. Se podría mejorar más el sitio colocando el formulario de búsqueda en cada página (esto es, en la plantilla base). Dejaremos esto de tarea para el hogar.

A continuación veremos un ejemplo más complejo. Pero antes de hacerlo, discutamos un tópico más abstracto: el "formulario perfecto".

7.2. El "formulario perfecto"

Los formularios pueden ser a menudo una causa importante de frustración para los usuarios de tu sitio. Consideremos el comportamiento de un hipotético formulario perfecto:

- Debería pedirle al usuario cierta información, obviamente. La accesibilidad y la usabilidad importan aquí. Así que es importante el uso inteligente del elemento `<label>` de HTML, y también lo es proporcionar ayuda contextual útil.
- Los datos suministrados deberían ser sometidos a una validación extensiva. La regla de oro para la seguridad de una aplicación web es "nunca confíes en la información que ingresa". Así que la validación es esencial.
- Si el usuario ha cometido algún error, el formulario debería volver a mostrarse, junto a mensajes de error detallados e informativos. 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.

¡Construir el formulario perfecto pareciera llevar mucho trabajo! Por suerte, el *framework* de formularios de Django está diseñado para hacer la mayor parte del trabajo por ti. Se le proporciona una descripción de los campos del formulario, reglas de validación, y una simple plantilla, y Django hace el resto. El resultado es un "formulario perfecto" que requiere de muy poco esfuerzo.

7.3. Creación de un formulario para comentarios

La mejor forma de construir un sitio que la gente ame es atendiendo a sus comentarios. Muchos sitios parecen olvidar esto; ocultan los detalles de su contacto en *FAQs*, y parecen dificultar lo más posible el encuentro con las personas.

Cuando tu sitio tiene millones de usuarios, esto puede ser una estrategia razonable. En cambio, cuando intentas formarte una audiencia, deberías pedir comentarios cada vez que se presente la oportunidad. Escribamos entonces un simple formulario para comentarios, y usémoslo para ilustrar al *framework* de Django en plena acción.

Comenzaremos agregando `(r'^contact/$', 'mysite.books.views.contact')` al URLconf, y luego definamos nuestro formulario. Los formularios en Django se crean de una manera similar a los modelos: declarativamente, empleando una clase de Python. He aquí la clase para nuestro simple formulario. Por convención, lo insertaremos en un nuevo archivo `forms.py` dentro del directorio de nuestra aplicación:

```
from django import newforms as forms

TOPIC_CHOICES = (
    ('general', 'General enquiry'),
    ('bug', 'Bug report'),
    ('suggestion', 'Suggestion'),
)

class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField()
    sender = forms.EmailField(required=False)
```

> **¿*"New" Forms?* ¿Qué?**
>
> Cuando Django fue lanzado al público por primera vez, poseía un sistema de formularios complicado y confuso. Como hacía muy dificultosa la producción de formularios, fue rescrito y ahora se llama *"newforms"* (nuevos formularios). Sin embargo, como todavía hay cierta cantidad de código que depende del "viejo" sistema de formularios, Django actualmente viene con ambos paquetes. Al momento de escribir ese libro, el viejo sistema de formularios de Django sigue disponible como `django.forms`, y el nuevo paquete como `django.newforms`. En algún momento esto va a cambiar, y `django.forms` hará referencia al nuevo paquete de formularios. Sin embargo, para estar seguros de que los ejemplos de este libro funcionen lo más ampliamente posible, todos harán referencia a `django.newforms`.

Un formulario de Django es una subclase de `django.newforms.Form`, tal como un modelo de Django es una subclase de `django.db.models.Model`. El módulo `django.newforms` también contiene cierta cantidad de clases `Field` para los campos. Una lista completa de éstas últimas se encuentra disponible en la documentación de Django, en http://www.djangoproject.com/documentation/0.96/newforms/.

Nuestro `ContactForm` consiste de tres campos: un tópico, que se puede elegir entre tres opciones; un mensaje, que es un campo de caracteres; y un emisor, que es un campo de correo electrónico y es opcional (porque incluso los comentarios anónimos pueden ser útiles). Hay una cantidad de otros tipos de campos disponibles, y puedes escribir nuevos tipos si ninguno cubre tus necesidades.

El objeto formulario sabe cómo hacer una cantidad de cosas útiles por sí mismo. Puede validar una colección de datos, puede generar sus propios *"widgets"* de HTML, puede construir un conjunto de mensajes de error útiles. Y si estás en perezoso, puede incluso dibujar el formulario completo por ti. Incluyamos esto en una vista y veámoslo en acción. En `views.py`:

```
from django.db.models import Q
from django.shortcuts import render_to_response
from models import Book
from forms import ContactForm

def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })
```

```
def contact(request):
    form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

y en `contact.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    <form action="." method="POST">
        <table>
            {{ form.as_table }}
        </table>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

La línea más interesante aquí es `{{ form.as_table }}`. `form` es nuestra instancia de ContactForm, que fue pasada al `render_to_response`. `as_table` es un método de ese objeto que reproduce el formulario como una secuencia de renglones de una tabla (también pueden usarse `as_ul` y `as_p`). El HTML generado se ve así:

```
<tr>
    <th><label for="id_topic">Topic:</label></th>
    <td>
        <select name="topic" id="id_topic">
            <option value="general">General enquiry</option>
            <option value="bug">Bug report</option>
            <option value="suggestion">Suggestion</option>
        </select>
    </td>
</tr>
<tr>
    <th><label for="id_message">Message:</label></th>
    <td><input type="text" name="message" id="id_message" /></td>
</tr>
<tr>
    <th><label for="id_sender">Sender:</label></th>
    <td><input type="text" name="sender" id="id_sender" /></td>
</tr>
```

Observa que las etiquetas `<table>` 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".

Nuestro formulario actualmente utiliza un *widget* `<input type="text">` para el campo del mensaje. Pero no queremos restringir a nuestros usuarios a una sola línea de texto, así que la cambiaremos por un *widget* `<textarea>`:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)
```

El *framework* de formularios divide la lógica de presentación para cada campo, en un conjunto de *widgets*. Cada tipo de campo tiene un *widget* por defecto, pero puedes sobreescribirlo fácilmente, o proporcionar uno nuevo de tu creación.

Por el momento, si se suministra el formulario, no sucede nada. Agreguemos nuestras reglas de validación:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

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 *unbound* no tiene datos asociados y simplemente sabe cómo representarse a sí mismo.

Intenta hacer clic en *Submit* en el formulario vacío. La página se volverá a cargar, mostrando un error de validación que informa que nuestro campo de mensaje es obligatorio.

Intenta también ingresar una dirección de correo electrónico inválida. El `EmailField` sabe cómo validar estas direcciones, por lo menos a un nivel razonable.

Cómo especificar datos iniciales

Al pasar datos directamente al constructor del formulario, estos se vinculan, y se indica que la validación debe ser efectuada. A menudo, necesitamos mostrar un formulario inicial con algunos campos previamente rellenados -- por ejemplo, en un formulario "editar". Podemos hacerlo con el argumento de palabras claves `initial`:

```
form = CommentForm(initial={'sender': 'user@example.com'})
```

Si nuestro formulario *siempre* usará los mismos valores por defecto, podemos configurarlos en la definición misma del formulario:

```
message = forms.CharField(widget=forms.Textarea(),
                          initial="Replace with your feedback")
```

7.4. Procesamiento de los datos suministrados

Una vez que el usuario ha llenado el formulario al punto de que pasa nuestras reglas de validación, necesitamos hacer algo útil con los datos. En este caso, deseamos construir un correo electrónico que contenga los comentarios del usuario, y enviarlo. Para esto, usaremos el paquete de correo electrónico de Django.

Pero antes, necesitamos saber si los datos son en verdad válidos, y si lo son, necesitamos una forma de accederlos. El *framework* de formularios hace más que validar los datos, también los convierte a tipos de datos de Python. Nuestro formulario para comentarios sólo trata con texto, pero si estamos usando campos como `IntegerField` o `DateTimeField`, el *framework* de formularios se encarga de que se devuelvan como un valor entero de Python, o como un objeto `datetime`, respectivamente.

Para saber si un formulario está vinculado (*bound*) a datos válidos, llamamos al método `is_valid()`:

```
form = ContactForm(request.POST)
if form.is_valid():
    # Process form data
```

Ahora necesitamos acceder a los datos. Podríamos sacarlos directamente del `request.POST`, pero si lo hiciéramos, no nos estaríamos beneficiando de la conversión de tipos que realiza el *framework* de formularios. En cambio, usamos `form.clean_data`:

```
if form.is_valid():
    topic = form.clean_data['topic']
    message = form.clean_data['message']
    sender = form.clean_data.get('sender', 'noreply@example.com')
    # ...
```

Observa que dado que `sender` no es obligatorio, proveemos un valor por defecto por si no fue proporcionado. Finalmente, necesitamos registrar los comentarios del usuario. La manera más fácil de hacerlo es enviando un correo electrónico al administrador del sitio. Podemos hacerlo empleando la función:

```
from django.core.mail import send_mail

# ...

send_mail(
    'Feedback from your site, topic: %s' % topic,
    message, sender,
    ['administrator@example.com']
)
```

La función `send_mail` tiene cuatro argumentos obligatorios: el asunto y el cuerpo del mensaje, la dirección del emisor, y una lista de direcciones destino. `send_mail` es un código conveniente que envuelve a la clase `EmailMessage` de Django. Esta clase provee características avanzadas como adjuntos, mensajes multiparte, y un control completo sobre los encabezados del mensaje.

Una vez enviado el mensaje con los comentarios, redirigiremos a nuestro usuario a una página estática de confirmación. La función de la vista finalizada se ve así:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.core.mail import send_mail
from forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            topic = form.clean_data['topic']
            message = form.clean_data['message']
            sender = form.clean_data.get('sender', 'noreply@example.com')
            send_mail(
                'Feedback from your site, topic: %s' % topic,
                message, sender,
                ['administrator@example.com']
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

Redirigir luego del POST

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 ayudar a prevenir este escenario. Así que luego de que se haya procesado el POST con éxito, redirige al usuario a otra página en lugar de retornar HTML directamente.

7.5. Nuestras propias reglas de validación

Imagina que hemos lanzado al público a nuestro formulario de comentarios, y los correos electrónicos han empezado a llegar. Nos encontramos con un problema: algunos mensajes vienen con sólo una o dos palabras, es poco probable que tengan algo interesante. Decidimos adoptar una nueva póliza de validación: cuatro palabras o más, por favor.

Hay varias formas de insertar nuestras propias validaciones 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 a la clase del formulario.

En este caso, necesitamos validación adicional sobre el campo `message`, así que debemos agregar un método `clean_message` a nuestro formulario:

```
class ContactForm(forms.Form):
```

```
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)

    def clean_message(self):
        message = self.clean_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Este nuevo método será llamado después del validador que tiene el campo por defecto (en este caso, el validador de un `CharField` obligatorio). Dado que los datos del campo ya han sido procesados parcialmente, necesitamos obtenerlos desde el diccionario `clean_data` del formulario.

Usamos una combinación de `len()` y `split()` para contar la cantidad de palabras. Si el usuario ha ingresado muy pocas palabras, lanzamos un error `ValidationError`. El texto que lleva esta excepción se mostrará al usuario como un elemento de la lista de errores.

Es importante que retornemos explícitamente el valor 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 será perdido.

7.6. Una presentación personalizada

La forma más rápida de personalizar la presentación de un formulario es mediante CSS. En particular, la lista de errores puede dotarse de mejoras visuales, y el elemento `<ul>` tiene asignada la clase `errorlist` para ese propósito. El CSS a continuación hace que nuestros errores salten a la vista:

```
<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>
```

Si bien es conveniente que el HTML del formulario sea generado por nosotros, en muchos casos la disposición por defecto no quedaría 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 sobreescrito, 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 }}`. Cualquier error asociado con un campo está disponible como `{{ form.fieldname.errors }}`. Podemos usar estas variables para construir nuestra propia plantilla para el formulario:

```
<form action="." method="POST">
    <div class="fieldWrapper">
        {{ form.topic.errors }}
        <label for="id_topic">Kind of feedback:</label>
        {{ form.topic }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
```

```
            {{ form.message }}
        </div>
        <div class="fieldWrapper">
            {{ form.sender.errors }}
            <label for="id_sender">Your email (optional):</label>
            {{ form.sender }}
        </div>
        <p><input type="submit" value="Submit"></p>
    </form>
```

`{{ form.message.errors }}` se muestra como un `<ul class="errorlist">` si se presentan errores y como 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.message.errors` como a un booleano o incluso iterar sobre la misma como en una lista, por ejemplo:

```
    <div class="fieldWrapper{% if form.message.errors%} errors{% endif%}">
        {% if form.message.errors%}
            <ol>
            {% for error in form.message.errors%}
                <li><strong>{{ error|escape }}</strong></li>
            {% endfor%}
            </ol>
        {% endif%}
        {{ form.message }}
    </div>
```

En caso de que hubieran errores de validación, se agrega la clase "errors" al `<div>` contenedor y se muestran los errores en una lista ordenada.

7.7. Creando formularios a partir de Modelos

Construyamos algo un poquito más interesante: un formulario que suministre los datos de un nuevo publicista a nuestra aplicación de libros del Capítulo 5.

Una regla de oro que es importante en el desarrollo de software, a la que Django intenta adherirse, es: no te repitas (del inglés *Don't Repeat Yourself*, abreviado DRY). Andy Hunt y Dave Thomas la definen como sigue, en *The Pragmatic Programmer*:

> Cada pieza de conocimiento debe tener una representación única, no ambigua, y de autoridad, dentro de un sistema.

Nuestro modelo de la clase `Publisher` dice que un publicista tiene un nombre, un domicilio, una ciudad, un estado o provincia, un país, y un sitio web. Si duplicamos esta información en la definición del formulario, estaríamos quebrando la regla anterior. En cambio, podemos usar este útil atajo: `form_for_model()`:

```
from models import Publisher
from django.newforms import form_for_model

PublisherForm = form_for_model(Publisher)
```

`PublisherForm` es una subclase de `Form`, tal como la clase `ContactForm` que creamos manualmente con anterioridad. Podemos usarla de la misma forma:

```
from forms import PublisherForm

def add_publisher(request):
    if request.method == 'POST':
        form = PublisherForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/add_publisher/thanks/')
    else:
        form = PublisherForm()
    return render_to_response('books/add_publisher.html', {'form': form})
```

El archivo `add_publisher.html` es casi idéntico a nuestra plantilla `contact.html` original, así que la omitimos. Recuerda además agregar un nuevo patrón al URLconf: `(r'^add_publisher/$', 'mysite.books.views.add_publisher')`.

Ahí se muestra un atajo más. Dado que los formularios derivados de modelos se emplean a menudo para guardar nuevas instancias del modelo en la base de datos, la clase del formulario creada por `form_for_model` incluye un conveniente método `save()`. Este método trata con el uso común; pero puedes ignorarlo si deseas hacer algo más que tenga que ver con los datos suministrados.

`form_for_instance()` es un método que está relacionado con el anterior, y puede crear formularios preinicializados a partir de la instancia de un modelo. Esto es útil al crear formularios "editar".

7.8. ¿Qué sigue?

Este capítulo concluye con el material introductorio de este libro. Los próximos trece capítulos tratan con varios tópicos avanzados, incluyendo la generación de contenido que no es HTML (Capítulo 11), seguridad (**'Capítulo 19'_**), y entrega del servicio (Capítulo 20).

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 vayas necesitando.

Comenzaremos el Capítulo 8 yendo hacia atrás, volviendo para darle una mirada más de cerca a las vistas y a los URLconfs (introducidos por primera vez en el Capítulo 3).

Capítulo 8

Vistas avanzadas y URLconfs

En el Capítulo 3, explicamos las bases de las funciones vista de Django y las URLconfs. Este capítulo entra en detalle sobre funcionalidad avanzada en esas dos partes del framework.

8.1. 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 varias maneras, como se describe las secciones que siguen.

8.1.1. Importación de funciones de forma efectiva

Considera esta URLconf, que se basa en el ejemplo del Capítulo 3:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead, hours_behind, now_in_chicago, now_in_londo

urlpatterns = patterns('',
    (r'^now/$', current_datetime),
    (r'^now/plus(\d{1,2})hours/$', hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', hours_behind),
    (r'^now/in_chicago/$', now_in_chicago),
    (r'^now/in_london/$', now_in_london),
)
```

Como se explicó en el Capítulo 3, cada entrada de la URLconf incluye su función vista asociada, que se pasa directamente como un método. Esto significa que es necesario importar las funciones view en la parte superior del módulo.

Pero a medida que las aplicaciones Django crecen en complejidad, sus URLconf crecen también, y 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. Este ejemplo de URLconf es equivalente al anterior:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^now/$', views.current_datetime),
    (r'^now/plus(\d{1,2})hours/$', views.hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', views.hours_behind),
    (r'^now/in_chicago/$', views.now_in_chicago),
    (r'^now/in_london/$', views.now_in_london),
)
```

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 de la función en lugar del método. Continuando con el ejemplo:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^now/$', 'mysite.views.current_datetime'),
    (r'^now/plus(\d{1,2})hours/$', 'mysite.views.hours_ahead'),
    (r'^now/minus(\d{1,2})hours/$', 'mysite.views.hours_behind'),
    (r'^now/in_chicago/$', 'mysite.views.now_in_chicago'),
    (r'^now/in_london/$', 'mysite.views.now_in_london'),
)
```

(Nota que los nombres de las vistas están entre comillas. Estamos usando `'mysite.views.current_datetime'` -- con comillas -- en lugar de `mysite.views.current_datetime`.)

Al usar esta técnica ya no es necesario importar las funciones vista; Django importa automáticamente la función vista apropiada la primera vez que sea necesaria, según el string que describe el nombre y la ruta de la función vista.

Otro atajo que puedes tomar al usar la técnica del string es sacar factor común de "prefijos view". En nuestro ejemplo URLconf, cada uno de los strings vista comienza con `'mysite.views'`, lo cual es redundante. Podemos factorizar ese prefijo común y pasarlo como primer argumento de `patterns()`, así:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^now/$', 'current_datetime'),
    (r'^now/plus(\d{1,2})hours/$', 'hours_ahead'),
    (r'^now/minus(\d{1,2})hours/$', 'hours_behind'),
    (r'^now/in_chicago/$', 'now_in_chicago'),
    (r'^now/in_london/$', 'now_in_london'),
)
```

Nota que no se pone un punto detrás del prefijo, ni un punto delante de los string vista. Django los pone automáticamente.

Con estos dos enfoques en mente, ¿cuál es mejor? Realmente depende de tu estilo personal de programación y tus necesidades.

Las siguientes son ventajas del enfoque string:

- Es más compacto, porque no requiere que importes las funciones vista.
- Resulta en URLconfs más fáciles de leer y de manejar si tus funciones vista están extendidas por varios módulos Python diferentes.

Las siguientes son ventajas del enfoque del método:

- Permite un fácil "empaquetado" de funciones vista. Ver la sección "**'Empaquetado de funciones vista'_**" más adelante en este capítulo.
- Es más "Pythónico" -- es decir, está más en línea con las tradiciones Python, como la de pasar funciones como objetos.

Ambos enfoques son válidos e incluso puedes mezclarlos dentro de la misma URLconf. La elección es tuya.

8.1.2. Usar múltiples prefijos de vista

En la práctica, si usas la técnica del string, probablemente termines mezclando vistas hasta el punto en que las vistas de tu URLconf no tengan un prefijo común. Sin embargo, todavía puedes sacar provecho del atajo del prefijo de las vistas para remover esta duplicación. Simplemente junta los objetos `patterns()`, así:

Antes:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^/?$', 'mysite.views.archive_index'),
    (r'^(\d{4})/([a-z]{3})/$', 'mysite.views.archive_month'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

Después:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^/?$', 'archive_index'),
    (r'^(\d{4})/([a-z]{3})/$', 'archive_month'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

Lo único que le importa al framework es que existe una variable a nivel módulo llamada `urlpatterns`. Esta variable puede ser construida de forma dinámica, como lo hacemos en este ejemplo.

8.1.3. Casos especiales de URLs en modo Debug

Hablando de construir `urlpatterns` de forma dinámica, quizás quieras aprovechar esta técnica para alterar el comportamiento de tu URLconf mientras estás en el modo depuración de Django. Para hacer eso simplemente comprueba el valor de la configuración `DEBUG` en tiempo de ejecución, así:

```
from django.conf.urls.defaults import*
from django.conf import settings

urlpatterns = patterns('',
    (r'^$', 'mysite.views.homepage'),
    (r'^(\d{4})/([a-z]{3})/$', 'mysite.views.archive_month'),
)

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^debuginfo$', 'mysite.views.debug'),
    )
```

En este ejemplo, la URL `/debuginfo/` sólo estará disponible si tu configuración `DEBUG` tiene el valor `True`.

8.1.4. 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 de palabra clave o argumentos posicionales -- y, en algunos casos, los dos al mismo tiempo. En una llamada por argumentos de palabra 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 sell(item, price, quantity):
    print "Selling%s unit(s) of%s at%s"% (quantity, item, price)
```

Para llamarla con argumentos posicionales, se especifican los argumentos en el orden en que están listados en la definición de la función:

```
sell('Socks', '$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:

```
sell(item='Socks', price='$2.50', quantity=6)
sell(item='Socks', quantity=6, price='$2.50')
sell(price='$2.50', item='Socks', quantity=6)
sell(price='$2.50', quantity=6, item='Socks')
sell(quantity=6, item='Socks', price='$2.50')
sell(quantity=6, price='$2.50', item='Socks')
```

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:

```
sell('Socks', '$2.50', quantity=6)
sell('Socks', price='$2.50', quantity=6)
sell('Socks', quantity=6, price='$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:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', views.year_archive),
    (r'^articles/(\d{4})/(\d{2})/$', views.month_archive),
)
```

Aquí está la misma URLconf, reescrita para usar grupos con nombre:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(?P<year>\d{4})/$', views.year_archive),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', views.month_archive),
)
```

Esto produce exactamente el mismo resultado que el ejemplo anterior, con una sutil diferencia: se le pasa a las funciones vista los valores capturados como argumentos clave en lugar de argumentos posicionales.

Por ejemplo, con los grupos sin nombre una petición a `/articles/2006/03/` resultaría en una llamada de función equivalente a esto:

```
month_archive(request, '2006', '03')
```

Sin embargo, con los grupos con nombre, la misma petición resultaría en esta llamada de función:

```
month_archive(request, year='2006', month='03')
```

En la práctica, usar grupos con nombres hace que tus URLconfs sean un poco más explícitas y menos propensas a bugs 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 `month_archive`. 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.

8.1.5. Comprender el algoritmo de combinación/agrupación

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. 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.

8.1.6. 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.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

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 comprobando la URL dentro de la vista para determinar la plantilla, como mostramos a continuación:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^(foo)/$', views.foobar_view),
    (r'^(bar)/$', views.foobar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

Sin embargo, el problema con esa solución es que acopla fuertemente tus URLs y tu código Si decides renombrar `/foo/` a `/fooey/`, 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.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foobar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foobar_view, {'template_name': 'template2.html'}),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

Como puedes ver, la URLconf en este ejemplo especifica `template_name` en la URLconf. La función vista lo trata como a cualquier otro parámetro.

Esta técnica de la opción extra en la URLconf es una linda 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 el sistema de vistas genéricas, que tratamos en el Capítulo 9.

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 particular, con URLs tales como:

```
/mydata/jan/01/
/mydata/jan/02/
/mydata/jan/03/
# ...
/mydata/dec/30/
/mydata/dec/31/
```

Esto es lo suficientemente simple de manejar -- puedes capturar los mismos en una URLconf como esta (usando sintaxis de grupos con nombre):

```
urlpatterns = patterns('',
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

Y la ***signature*** de la función vista se vería así:

```
def my_view(request, month, day):
    # ....
```

Este enfoque es simple y directo -- no es nada que no hayamos visto antes. El truco entra en juego cuando quieres agregar otra URL que usa `my_view` pero cuya URL no incluye un `month` ni/o un `day`.

Por ejemplo, podrías querer agregar otra URL, `/mydata/birthday/`, que sería equivalente a `/mydata/jan/06/`. Puedes sacar provecho de opciones extra de las URLconf de la siguiente forma:

```
urlpatterns = patterns('',
    (r'^mydata/birthday/$', views.my_view, {'month': 'jan', 'day': '06'}),
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

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 `month` y `day` -- no importa si los mismos provienen de la captura de la URL o de parámetros extra.

Convirtiendo una vista en genérica

Es una buena práctica de programación el "factorizar" para aislar las partes comunes del código. Por ejemplo, con estas dos funciones Python:

```
def say_hello(person_name):
    print 'Hello,%s'% person_name

def say_goodbye(person_name):
    print 'Goodbye,%s'% person_name
```

podemos extraer el saludo para convertirlo en un parámetro:

```
def greet(person_name, greeting):
    print '%s,%s'% (greeting, person_name)
```

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 nivel más alto de tus vistas. En lugar de pensar "Esta vista muestra una lista de objetos `Event`" y "Esta otra vista muestra una lista de objetos `BlogEntry`", 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.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^events/$', views.event_list),
```

```
    (r'^blog/entries/$', views.entry_list),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import Event, BlogEntry

def event_list(request):
    obj_list = Event.objects.all()
    return render_to_response('mysite/event_list.html', {'event_list': obj_list})

def entry_list(request):
    obj_list = BlogEntry.objects.all()
    return render_to_response('mysite/blogentry_list.html', {'entry_list': obj_list})
```

Ambas vistas hacen esencialmente lo mismo: muestran una lista de objetos. Refactoricemos el código para extraer el tipo de objetos que muestran:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import models, views

urlpatterns = patterns('',
    (r'^events/$', views.object_list, {'model': models.Event}),
    (r'^blog/entries/$', views.object_list, {'model': models.BlogEntry}),
)

# views.py

from django.shortcuts import render_to_response

def object_list(request, model):
    obj_list = model.objects.all()
    template_name = 'mysite/%s_list.html'% model.__name__.lower()
    return render_to_response(template_name, {'object_list': obj_list})
```

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 una listado de objetos, podemos simplemente reusar esta vista `object_list` en lugar de escribir código de vista. 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 ULconf puede pasar cualquier tipo de objetos Python -- no sólo strings.
- 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 `object_list`. Podemos fácilmente cambiar este nombre de variable a `blogentry_list` o `event_list`, 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 ocupamos de las vistas genéricas incluidas con Django en el próximo capítulo.

Pasando opciones de configuración a una vista

Si estás distribuyendo una aplicación Django, es probable que tus usuarios deseen 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 my_view(request, template_name):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

Entendiendo la precedencia entre valores captuardos vs. opciones extra

Cuando se presenta un conflicto, los parámetros extra de la URLconf tiene 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.defaults import *

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id': 3}),
)
```

Aquí, tanto la expresión regular como el diccionario extra incluye un `id`. Tiene precedencia el `id` fijo especificado. Esto significa que cualquier petición (por ej. `/mydata/2/` o `/mydata/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 ese error.

8.1.7. Usando argumentos de vista por omisión

Otro truco cómodo 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.defaults import *

urlpatterns = patterns('',
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    # ...
```

Aquí, ambos patrones de URL apuntan a la misma vista -- `views.page` -- pero el primer patrón no captura nada de la URL. Si el primer patrón es disparado, la función `page()` usará su argumento por omisión para `num`, `"1"`. Si el segundo patrón es disparado, `page()` 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 `template_name`:

```
def my_view(request, template_name='mysite/my_view.html'):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

8.1.8. 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 = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

Esto se disparará con URLs como `/myblog/entries/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 add_stage(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: Coloca lógica de URLs en la vista. Una manera más elegante sería la de hacer uso del hecho que las URLconfs se procesan desde arriba hacia abajo:

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', 'django.contrib.admin.views.auth.user_add_stage'),
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

Con esto, una petición de `/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).

8.1.9. 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:

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

el argumento `year` de `views.year.archive()` 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 una vista, este error se vería así:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/(\d{4})/(\d{2})/(\d{2})/$', views.day_archive),
)

# views.py

import datetime

def day_archive(request, year, month, day)
    # The following statement raises a TypeError!
    date = datetime.date(year, month, day)
```

En cambio `day_archive` puede ser escrito correctamente de la siguiente forma:

```
def day_archive(request, year, month, day)
    date = datetime.date(int(year), int(month), int(day))
```

Notar 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.

8.1.10. 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 de `http://www.example.com/myapp/` Django tratará de encontrar una coincidencia para `myapp/`. En una petición de `http.//www.example.com/myapp/?page3` Django tratará de buscar una coincidencia para `myapp/`.

El método de la petición (por ej. `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 el manejar de maneras distintas en base al método de la petición.

8.2. 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.

Tu 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.defaults import *

urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

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 `mysite.blog.urls`:

```
from django.conf.urls.defaults import *
```

```
urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

Con esas dos URLconfs, veremos aquí cómo serían manejadas algunas peticiones de ejemplo:

- `/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 `mysite.blog.urls`.
- `/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 la líneas en la URLconf `mysite.blog.urls`.
- `/about/`: Esto coincide con el patrón de la vista `mysite.views.about` en la primera URLconf, demostrando que puedes combinar patrones `include()` con patrones no `include()`.

8.2.1. Cómo trabajan los parámetros capturados con include()

Una URLconf incluida recibe todo parámetro que se haya capturado desde las URLconf padres, por ejemplo:

```
# root urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# foo/urls/blog.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'foo.views.blog_index'),
    (r'^archive/$', 'foo.views.blog_archive'),
)
```

En este ejemplo, la variable capturada `username()` es pasada a la URLconf incluida y, por lo tanto, a *todas* las funciones vista en dicha URLconf.

Notar que los parámetros capturados serán pasados *siempre* a *todas* las líneas en la URLconf incluida, con independencia de si la vista de la línea realmente acepta esos 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 estás pasando.

8.2.2. 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.

Conjunto uno:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)
```

```
# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

Conjunto dos:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
    (r'^rss/$', 'mysite.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 líneas en la URLconf incluida, sin importar de si la vista de la línea realmente acepta esas opciones como válidas. Por eta razón esta técnica es útil sólo si estás seguro que todas las vistas en la URLconf incluida acepta las opciones extra que estás pasando.

8.3. ¿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 formas en las cuales se puede reducir el código de tus vistas y URLconfs.

El próximo paso lógico en la reducción de código es eliminar completamente la necesidad de escribir vistas. Ese es el tópico del **'próximo capítulo'_**.

Duplicate explicit target name: "próximo capítulo".

Capítulo 9

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* de Django fueron desarrolladas para aliviar ese dolor. Éstas 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.

El Capítulo 8 refirió brevemente sobre cómo harías para crear una vista "genérica". Para repasar, podemos reconocer ciertas tareas comunes, como mostrar una lista de objetos, y escribir código que muestra una lista 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 para hacer lo siguiente:

- Realizar tareas "sencillas" comunes: redirigir a una página diferente y renderizar una plantilla dada.
- Mostrar páginas de listado y detalle para un solo objeto. Las vistas `event_list` y `entry_list` del Capítulo 8 son ejemplos de vistas de listado. Una página de evento simple 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 Weblog 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 para realizar las tareas más comunes que encuentran los desarrolladores.

9.1. Usar vistas genéricas

Todas estas vistas se usan creando diccionarios de configuración en tus archivos URLconf y pasando estos diccionarios como el tercer miembro de la tupla URLconf para un patrón dado.

Por ejemplo, ésta es una URLconf simple que podrías usar para presentar una página estática "about" (acerca de):

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    })
)
```

Aunque esto podría verse un poco "mágico" a primera vista -- ¡mira, una vista sin código! --, es en realidad exactamente lo mismo que los ejemplos en el la vista `direct_to_template` simplemente toma información del diccionario de parámetros extra y usa esa información cuando renderiza la vista.

Ya que esta vista genérica -- y todas las otras -- es una función de vista regular como cualquier otra, podemos reusarla dentro de nuestras propias vistas. Como ejemplo, extendamos nuestro ejemplo "about" para mapear URLs de la forma `/about/<cualquiercosa>/` para renderizar estáticamente `/about/<cualquiercosa>.html`. Haremos esto primero modificando la URLconf para que apunte a una función de vista:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
from mysite.books.views import about_pages

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    ('^about/(w+)/$', about_pages),
)
```

A continuación, escribimos la vista `about_pages`:

```
from django.http import Http404
from django.template import TemplateDoesNotExist
from django.views.generic.simple import direct_to_template

def about_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html"% page)
    except TemplateDoesNotExist:
        raise Http404()
```

Aquí estamos tratando `direct_to_template` como cualquier otra función. Ya que esta devuelve una `HttpResponse`, podemos retornarlo así como está. La única ligera dificultad aquí es ocuparse de las plantillas perdidas. No queremos que una plantilla inexistente cause un error de servidor, por lo tanto atrapamos las excepciones `TemplateDoesNotExist` y en su lugar devolvemos errores 404.

¿Hay una vulnerabilidad de seguridad aquí?

Los lectores atentos pueden haber notado un posible agujero de seguridad: estamos construyendo el nombre de la plantilla usando contenido interpolado proveniente del navegador (`template="about/%s.html"% page`). A primera vista, esto parece como una clásica vulnerabilidad de *recorrido de directorio* [11] (discutida en detalle en el **'Capítulo 19'_**). ¿Pero es realmente una vulnerabilidad?

No exactamente. Sí, un valor creado maliciosamente de page podría causar un recorrido de directorio, pero aunque page *es* tomado de la URL solicitada, no todos los valores serán aceptados. La clave esta en la URLconf: estamos usando la expresión regular `\w+` para verificar la parte page de la URL y `\w` sólo acepta letras y números. Por lo tanto, cualquier caracter malicioso (puntos y barras, en este caso) serán rechazadas por el URL resolver antes de alcanzar la vista en sí.

9.2. Vistas genéricas de objetos

La vista genérica `direct_to_template` 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.

Demos un vistazo a una de estas vistas genéricas: la vista "object list". Usaremos el objeto `Publisher` del Capítulo 5:

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
```

```
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    class Meta:
        ordering = ["-name"]

    class Admin:
        pass
```

Para construir una página listado de todos los books, usaremos la URLconf bajo estas líneas:

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    "queryset" : Publisher.objects.all(),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

Ese es todo el código Python que necesitamos escribir. Sin embargo, todavía necesitamos escribir una plantilla. Podríamos decirle explícitamente a la vista `object_list` que plantilla debe usar incluyendo una clave `template_name` en el diccionario de argumentos extra, pero en la ausencia de una plantilla explícita Django inferirá una del nombre del objeto. En este caso, la plantilla inferida será `"books/publisher_list.html"` -- la parte "books" proviene del nombre de la aplicación que define el modelo, mientras que la parte "publisher" es sólo la versión en minúsculas del nombre del modelo.

Esta plantilla será renderizada en un contexto que contiene una variable llamada `object_list` la cual contiene todos los objetos book. Una plantilla muy simple podría verse como la siguiente:

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

{% block content%}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list%}
            <li>{{ publisher.name }}</li>
        {% endfor%}
    </ul>
{% endblock%}
```

Eso es realmente todo en lo referente al tema. Todas las geniales características de las vistas genéricas provienen de cambiar el diccionario "info" pasado a la vista genérica. El Apéndice D documenta todas las vistas genéricas y todas sus opciones en detalle; el resto de este capítulo considerará algunas de las maneras comunes en que tú puedes personalizar y extender las vistas genéricas.

9.3. 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 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.

9.3.1. Crear contextos de plantilla "amistosos"

Tal vez hayas notado que el ejemplo de la plantilla publisher list almacena todos los books en una variable llamada `object_list`. Aunque que esto funciona bien, no es una forma "amistosa" para los autores de plantillas: ellos sólo tienen que "saber" aquí que están trabajando con books. Un nombre mejor para esa variable sería `publisher_list`; el contenido de esa variable es bastante obvio.

Podemos cambiar el nombre de esa variable fácilmente con el argumento `template_object_name`:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

Proveer un `template_object_name` útil es siempre una buena idea. Tus compañeros de trabajo que diseñan las plantillas te lo agradecerán.

9.3.2. 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 otros publisher en cada página de detalle de un publisher. La vista genérica `object_detail` provee el publisher al contexto, pero parece que no hay forma de obtener una lista de *todos* los publishers en esa plantilla.

Pero sí la hay: todas las vistas genéricas toman un parámetro opcional extra, `extra_context`. Este es un diccionario de objetos extra que serán agregados al contexto de la plantilla. Por lo tanto, para proporcionar la lista de todos los publishers en la vista de detalles, usamos un diccionario info como el que sigue:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : Book.objects.all()}
}
```

Esto llenaría una variable `{{ book_list }}` en el contexto de la plantilla. Este patrón puede ser usado para pasar cualquier información hacia la plantilla para la vista genérica. Es muy práctico.

Sin embargo, en realidad hay un error sutil aquí -- ¿puedes detectarlo?

El problema aparece cuando las consultas en `extra_context` son evaluadas. Debido a que este ejemplo coloca `Publisher.objects.all()` en la URLconf, sólo se evaluará una vez (cuando la URLconf se cargue por primera vez). Una vez que agregues o elimines publishers, notarás que la vista genérica no refleja estos cambios hasta que reinicias el servidor Web (mira "Almacenamiento en caché y QuerySets" en el Apéndice C para mayor información sobre cuándo los QuerySets son almacenados en la cache y evaluados).

> **Nota**
>
> Este problema no se aplica al argumento queryset de las vistas genéricas. Ya que Django sabe que ese QuerySet en particular *nunca* debe ser almacenado en la caché, la vista genérica se hace cargo de limpiar la caché cuando cada vista es renderizada.

La solución es usar un callback [12] en `extra_context` en vez de un valor. Cualquier callable [13] (por ejemplo, una función) que sea pasado a `extra_context` será evaluado cuando su vista sea renderizada (en vez de sólo la primera vez). Puedes hacer esto con una función explícitamente definida:

```
def get_books():
    return Book.objects.all()

publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : get_books}
}
```

o puedes usar una versión menos obvia pero más corta que se basa en el hecho de que `Publisher.objects.all` es en sí un callable:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : Book.objects.all}
}
```

Nota la falta de paréntesis después de `Book.objects.all`; esto hace referencia a la función sin invocarla realmente (cosa que hará la vista genérica luego).

9.3.3. Mostrar subconjuntos de objetos

Ahora echemos un vistazo más de cerca a esta clave `queryset` que hemos venido usando hasta aquí. La mayoría de las vistas genéricas usan uno de estos argumentos `queryset` -- es la manera en que la vista conoce qué conjunto de objetos mostrar (mira "Seleccionando objetos" en el Capítulo 5 para una introducción a los QuerySets, y mira el Apéndice C para los detalles completos).

Para tomar un ejemplo simple, tal vez querríamos ordenar una lista de books por fecha de publicación, con el más reciente primero.

```
book_info = {
    "queryset" : Book.objects.all().order_by("-publication_date"),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/$', list_detail.object_list, book_info),
)
```

Este es un muy lindo y simple ejemplo, pero ilustra bien la idea. Por supuesto, tú usualmente querrás hacer más que sólo reordenar objetos. Si quieres presentar una lista de books de un publisher particular, puedes usar la misma técnica:

```
apress_books = {
    "queryset": Book.objects.filter(publisher__name="Apress Publishing"),
    "template_name" : "books/apress_list.html"
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/apress/$', list_detail.object_list, apress_books),
)
```

Nota que además de un `queryset` filtrado, 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" [14], que puede no ser lo que queremos.

También nota que ésta no es una forma muy elegante de hacer publisher-specific books. Si queremos agregar otra página publisher, necesitamos otro puñado de líneas en la URLconf, y más de unos pocos publishers no será razonable. Enfrentaremos este problema en la siguiente sección.

> **Nota**
>
> Si obtienes un error 404 cuando solicitas /books/apress/, para estar seguro, verifica que en realidad tienes un Publisher con el nombre 'Apress Publishing'. Las vistas genéricas tienen un parámetro `allow_empty` para este caso. Mira el Apéndice D para mayores detalles.

9.3.4. Filtrado complejo con funciones wrapper

Otra necesidad común es filtrar los objetos que se muestran en una página listado por alguna clave en la URLconf. Anteriormente codificamos [15] el nombre del publisher en la URLconf, pero ¿qué pasa si queremos escribir una vista que muestre todos los books por algún publisher arbitrario?. Podemos "encapsular" [16] la vista genérica `object_list` para evitar escribir mucho código a mano. Como siempre, empezamos escribiendo una URLconf.

```
urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/(w+)/$', books_by_publisher),
)
```

A continuación, escribiremos la vista `books_by_publisher`:

```
from django.http import Http404
from django.views.generic import list_detail
from mysite.books.models import Book, Publisher

def books_by_publisher(request, name):

    # Look up the publisher (and raise a 404 if it can't be found).
    try:
        publisher = Publisher.objects.get(name__iexact=name)
    except Publisher.DoesNotExist:
        raise Http404

    # Use the object_list view for the heavy lifting.
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(publisher=publisher),
        template_name = "books/books_by_publisher.html",
        template_object_name = "books",
        extra_context = {"publisher" : publisher}
    )
```

Esto funciona porque en realidad no hay nada en especial sobre las vistas genéricas -- son sólo funciones Python. Como cualquier función de vista, las vistas genéricas esperan un cierto conjunto de argumentos y retornan objetos `HttpResponse`. Por lo tanto, es increíblemente fácil encapsular una pequeña función sobre una vista genérica que realiza trabajo adicional antes (o después; mira la siguiente sección) de pasarle el control a la vista genérica.

> **Nota**
>
> Nota que en el ejemplo anterior pasamos el publisher que se está mostrando actualmente en el `extra_context`. Esto es usualmente una buena idea en wrappers de esta naturaleza; le permite a la plantilla saber qué objeto "padre" esta siendo navegado en ese momento.

9.3.5. 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 `last_accessed` en nuestro objeto `Author` que estuvimos usando para tener un registro de la última vez que alguien vio ese author. La vista genérica `object_detail`, 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 author en la URLconf para que apunte a una vista personalizada:

```
from mysite.books.views import author_detail

urlpatterns = patterns('',
    #...
    (r'^authors/(?P<author_id>d+)/$', author_detail),
)
```

Luego escribiremos nuestra función wrapper:

```
import datetime
from mysite.books.models import Author
```

```
from django.views.generic import list_detail
from django.shortcuts import get_object_or_404

def author_detail(request, author_id):
    # Look up the Author (and raise a 404 if she's not found)
    author = get_object_or_404(Author, pk=author_id)

    # Record the last accessed date
    author.last_accessed = datetime.datetime.now()
    author.save()

    # Show the detail page
    return list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )
```

> **Nota**
>
> Este código en realidad no funcionará a menos que agregues un campo `last_accessed` a tu modelo `Author` y agregues una plantilla `books/author_detail.html`.

Podemos usar un método similar para alterar la respuesta devuelta por la vista genérica. Si quisiéramos proporcionar una versión en texto plano [17] que se pueda descargar desde la lista de autores, podríamos usar una vista como esta:

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        request,
        queryset = Author.objects.all(),
        mimetype = "text/plain",
        template_name = "books/author_list.txt"
    )
    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

Esto funciona porque la vista genérica devuelve simplemente objetos `HttpResponse` que pueden ser tratados como diccionarios para establecer las cabeceras HTTP. Este arreglo de `Content-Disposition`, por otro lado, instruye al navegador a descargar y guardar la página en vez de mostrarla en pantalla.

9.4. ¿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. El Apéndice D cubre todas las vistas disponibles en detalle, y es de lectura obligada si quieres sacar el mayor provecho de esta característica.

En el **'próximo capítulo'_** ahondamos profundamente en el funcionamiento interno de las plantillas de Django, mostrando todas las maneras geniales en que pueden ser extendidas. Hasta ahora, hemos tratado el sistema de plantillas meramente como una herramienta estática que puedes usar para renderizar tu contenido.

Duplicate explicit target name: "próximo capítulo".

[11] N. del T.: directory traversal vulnerability.
[12] N. del T.: llamada a función.
[13] N. del T.: en Python cualquier objeto que puede ser llamado como función.
[14] N. del T.: "vanilla" object list.
[15] N. del T.: hard-coded.
[16] N. del T.: "wrap".
[17] N. del T.: en texto plano.

Capítulo 10

Extendiendo el sistema de plantillas

Aunque la mayor parte de tu interacción con el sistema de plantillas (*templates*) de Django será en el rol de autor, probablemente quieras 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.

Si estás tratando de utilizar el sistema de plantillas de Django como parte de otra aplicación, es decir, sin el resto del framework, lee la sección "Configurando el Sistema de plantillas en modo autónomo" luego en este mismo capítulo.

10.1. 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 %}
    Thanks for logging in!
  {% else %}
    Please log in.
  {% endif %}
  ```

- Una *variable* es un símbolo dentro de una plantilla que emite un valor.

  Las etiquetas de variable deben ser rodeadas por `{{` y `}}`:

  ```
  My first name is {{ first_name }}. My last name is {{ last_name }}.
  ```

- Un *contexto* es un mapa 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.

Para más detalles acerca de estos términos, referirse al Capítulo 4.

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.

10.2. Procesadores 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.

Usa `RequestContext` cuando no quieras especificar el mismo conjunto de variables una y otra vez en una serie de plantillas. Por ejemplo, considera estas cuatro vistas:

```
from django.template import loader, Context

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am view 1.'
    })
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the second view.'
    })
    return t.render(c)

def view_3(request):
# ...
    t = loader.get_template('template3.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the third view.'
    })
    return t.render(c)

def view_4(request):
    # ...
    t = loader.get_template('template4.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the fourth view.'
    })
    return t.render(c)
```

A propósito *no* hemos usado el atajo `render_to_response` 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 -- `app`, `user` y `ip_address` -- a su plantilla. ¿No sería bueno poder eliminar esa redundancia?

`RequestContext` y los **procesadores de contexto** fueron creado 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_to_response()`. El secreto está en utilizar `RequestContext` en lugar de `Context` cuando renderizamos 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):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = RequestContext(request, {'message': 'I am view 1.'},
            processors=[custom_proc])
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = RequestContext(request, {'message': 'I am the second view.'},
            processors=[custom_proc])
    return t.render(c)

def view_3(request):
    # ...
    t = loader.get_template('template3.html')
    c = RequestContext(request, {'message': 'I am the third view.'},
            processors=[custom_proc])
    return t.render(c)

def view_4(request):
    # ...
    t = loader.get_template('template4.html')
    c = RequestContext(request, {'message': 'I am the fourth view.'},
            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 contexto de la plantilla. Eso es todo lo que hace.
- Hemos cambiado las cuatro 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 *list* o tupla de funciones procesadoras de contexto a utilizar. En este caso, pasamos `custom_proc`, nuestro procesador de contexto definido previamente.
- Ya no es necesario en cada vista incluir `app`, `user` o `ip_address` 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 `message` es creada de manera diferente en cada una de las vistas.

En el Capítulo 4, presentamos el atajo `render_to_response()`, el cual nos ahorra tener que llamar a `loader.get_templat` luego crear un `Context` y ademas, 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_to_response()`, pero es posible -- y preferible -- utilizar los procesadores de contexto junto a `render_to_response()`. Esto lo logramos mediante el argumento `context_instance` de la siguiente manera:

```
from django.shortcuts import render_to_response
from django.template import RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    return render_to_response('template1.html',
        {'message': 'I am view 1.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_2(request):
    # ...
    return render_to_response('template2.html',
        {'message': 'I am the second view.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_3(request):
    # ...
    return render_to_response('template3.html',
        {'message': 'I am the third view.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_4(request):
    # ...
    return render_to_response('template4.html',
        {'message': 'I am the fourth view.'},
        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 usar procesadores de contexto no nos ahorra mucho código si tenemos que escribir `processors` constantemente.

Por esta razón, Django admite el uso de procesadores de contexto *globales*. El parámetro de configuración `TEMPLATE_CONTEXT_PROC` 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 *strings*, 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 numero de procesadores de contexto simples, entre ellos los que están activos por defecto.

10.2.1. 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 el Capítulo 12 encontrarás más información acerca de usuarios, permisos y mensajes.

10.2.2. django.core.context_processors.debug

Este procesador añade 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`.

10.2.3. 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 E se especifica más información sobre estos parámetros.

10.2.4. django.core.context_processors.request

Si este procesador está habilitado, cada `RequestContext` contendrá una variable `request`, la cual es el actual objeto `HttpRequest`. Este procesador no está habilitado por defecto.

10.2.5. 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.

10.3. 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 (*custom*) 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_name_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 esas funciones usan por omisión el valor de tu 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, por omisión, 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.

  Esto significa que puedes almacenar plantillas en tus aplicaciones individuales, facilitando la distribución de aplicaciones Django con plantillas por omisión. Por ejemplo si `INSTALLED_APPS` contiene `('myproject.polls', 'myproject.music')` entonces `get_template('foo.html')` buscará plantillas en el siguiente orden:

  - `/path/to/myproject/polls/templates/foo.html`
  - `/path/to/myproject/music/templates/foo.html`

  Notar que el cargador realiza una optimización cuando es importado por primera vez: hace caching de una lista de cuales de los paquetes en `INSTALLED_APPS` tienen un sub-directorio `templates`.

  Por omisión este cargador está activo.
- `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.

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.

10.4. Extendiendo 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 ensamblarás tus propias bibliotecas de etiquetas y filtros que se adapten a tus propias necesidades. Afortunadamente, es muy fácil definir tu propia funcionalidad.

10.4.1. Creando una biblioteca para plantillas

Ya sea que estés escribiendo etiquetas o filtros personalizados, la primera tarea a realizar es crear una **biblioteca para plantillas** -- un pequeño fragmento de infraestructura con el cual Django puede interactuar.

La creación de una biblioteca para plantillas es un proceso de dos pasos:

- Primero, decidir qué aplicación Django alojará la biblioteca. 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 cual de las dos rutas tomes, asegúrate de agregar la aplicación a tu variable de configuración `INSTALLED_APPS`. Explicaremos esto un poco más adelante.

- Segundo, crear un directorio `templatetags` en el paquete de aplicación Django apropiado. Debe encontrarse en el mismo nivel que `models.py`, `views.py`, etc. Por ejemplo:

  ```
  books/
      __init__.py
      models.py
      templatetags/
      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 `poll_extras.py`, entonces deberás escribir lo siguiente en una plantilla:

  ```
  {% load poll_extras %}
  ```

  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 plantillas 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 registradas 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`, usarás la misma para crear filtros y etiquetas para plantillas.

10.4.2. Escribiendo 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.

Esta es un ejemplo de definición de un filtro:

```
def cut(value, arg):
    "Removes all values of arg from the given string"
    return value.replace(arg, '')
```

Y este es un ejemplo de cómo se usaría:

```
{{ somevariable|cut:"0" }}
```

La mayoría de los filtros no reciben argumentos. En ese caso, basta con que no incluyas el argumento en tu función:

```
def lower(value): # Only one argument.
    "Converts a string into all lowercase"
    return value.lower()
```

Una vez que has escrito tu definición de filtro, necesitas registrarlo en tu instancia de `Library`, para que esté disponible para el lenguaje de plantillas de Django:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

El método `Library.filter()` tiene dos argumentos:

- El nombre del filtro (una cadena)
- La función filtro propiamente dicha

Si estás usando Python 2.4 o más reciente, puedes usar `register.filter()` como un decorador:

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.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 `cut`:

```
from django import template

register = template.Library()

@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')
```

10.4.3. Escribiendo 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 como 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`.

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. Por ende, 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 qué hace el método `render()` del nodo.

En las secciones que siguen, explicaremos todos los pasos necesarios para escribir una etiqueta propia.

Escribiendo 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` (ver `http://www.djangoproject.com/r/py` 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>The time is {% current_time "%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

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        msg = '%r tag requires a single argument' % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)
    return CurrentTimeNode(format_string[1:-1])
```

Hay muchas cosas en juego aquí:

- `parser` es la instancia del *parser*. No lo necesitamos en este ejemplo.
- `token.contents` es un *string* con los contenidos crudos de la etiqueta, en nuestro ejemplo sería: `'current_time "%Y-%m-%d%I:%M%p"'`.
- El método `token.split_contents()` separa los argumentos en sus espacios, mientras deja unidas a los *strings*. Evite utilizar `token.contentes.split()` (el cual usa la semántica natural de Python para dividir *strings*, 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 `CurrentTimeNode` (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 `Nodo`; cualquier otro valor es un error.

Escribiendo 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

class CurrentTimeNode(template.Node):

    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        now = datetime.datetime.now()
        return now.strftime(self.format_string)
```

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.

Registrando 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('current_time', do_current_time)
```

El método `tag()` toma dos argumentos:

- El nombre de la etiqueta de plantilla (*string*). Si esto se omite, se utilizará el nombre de la función de compilación.
- 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 2.4 o posterior:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(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.

Definiendo 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 `CurrentTimeNode` que define una variable de plantilla, `current_time`, en lugar de devolverla:

```
class CurrentTimeNode2(template.Node):

    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        now = datetime.datetime.now()
        context['current_time'] = now.strftime(self.format_string)
        return ''
```

Devolvemos un *string* vacío, 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 un *string* vacío.

De esta manera usaríamos esta nueva versión de nuestra etiqueta:

```
{% current_time2 "%Y-%M-%d%I:%M%p"%}
<p>The time is {{ current_time }}.</p>
```

Pero hay un problema con `CurrentTimeNode2`: el nombre de la variable `current_node` está definido dentro del código. Esto significa que tendrás que asegurar que `{{ current_time }}` no sea utilizado en otro lugar dentro de la plantilla, ya que `{% current_time%}` sobreescribirá el valor de esa otra variable.

Una solución más limpia, es poder recibir el nombre de la variable en la etiqueta de plantilla así:

```
{% get_current_time "%Y-%M-%d%I:%M%p" as my_current_time%}
<p>The current time is {{ my_current_time }}.</p>
```

Para hacer esto, necesitaremos modificar tanto la función de compilación como la clase `Node` de esta manera:

```
import re

class CurrentTimeNode3(template.Node):

    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name

    def render(self, context):
        now = datetime.datetime.now()
        context[self.var_name] = now.strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        msg = '%r tag requires arguments'% token.contents[0]
        raise template.TemplateSyntaxError(msg)

    m = re.search(r'(.*?) as (\w+)', arg)
    if m:
        fmt, var_name = m.groups()
    else:
        msg = '%r tag had invalid arguments'% tag_name
        raise template.TemplateSyntaxError(msg)

    if not (fmt[0] == fmt[-1] and fmt[0] in ('"', "'")):
```

```
        msg = "%r tag's argument should be in quotes"% tag_name
        raise template.TemplateSyntaxError(msg)

    return CurrentTimeNode3(fmt[1:-1], var_name)
```

Ahora, `do_current_time()` pasa el *string* de formato junto al nombre de la variable a `CurrentTimeNode3`.

Evaluando 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 la etiqueta estándar `{% 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.

Evaluando hasta otra etiqueta de bloque y guardando 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 %}
    This will appear in uppercase, {{ your_name }}.
{% endupper %}
```

Como en el ejemplo previo, utilizaremos `parser.parse()` pero esta vez pasamos el resultado en `nodelist` a `Node`:

```
@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 complejo, examina el código fuente para las etiquetas `{% if%}`, `{% for%}`, `{% ifequal%}` y `{% ifchanged%}`. Puedes encontrarlas en `django/template/defaulttags.py`.

10.4.4. 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 `current_time` 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 un 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 `current_time` podría entonces ser escrita de la siguiente manera:

```
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(current_time)
```

En Python 2.4 la sintaxis de decorador también funciona:

```
@register.simple_tag
def current_time(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 de 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.

10.4.5. 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 opciones para un simple objeto `Poll` con múltiples opciones. Usaremos una etiqueta como esta:

```
{% show_results poll%}
```

El resultado será algo como esto:

```
<ul>
  <li>First choice</li>
  <li>Second choice</li>
  <li>Third choice</li>
</ul>
```

Primero definimos la función que toma el 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 show_books_for_author(author):
    books = author.book_set.all()
    return {'books': books}
```

Luego creamos la plantilla usada para renderizar la salida de la etiqueta. Siguiendo con nuestro ejemplo, la plantilla es muy simple:

```
<ul>
{% for book in books %}
    <li> {{ book }} </li>
{% endfor %}
</ul>
```

Finalmente creamos y registramos la etiqueta de inclusión invocando el método `inclusion_tag()` sobre un objeto `Library`.

Continuando con nuestro ejemplo, si la plantilla se encuentra en un archivo llamado `polls/result_snippet.html`, registraremos la plantilla de la siguiente manera:

```
register.inclusion_tag('books/books_for_author.html')(show_books_for_author)
```

Como siempre, la sintaxis de decoradores de Python 2.4 también funciona, de manera que en cambio podríamos haber escrito:

```
@register.inclusion_tag('books/books_for_author.html')
def show_books_for_author(show_books_for_author):
    ...
```

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:

```
Jump directly to <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 %}
```

10.5. Escribiendo cargadores de plantillas personalizados

Los cargadores de plantillas incluidos con Django (descriptos 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_name, template_dirs=None)
```

El argumento `template_name` 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 contiendo plantillas:

```
import zipfile
from django.conf import settings
from django.template import TemplateDoesNotExist

def load_template_source(template_name, template_dirs=None):
    """Template loader that loads templates from a ZIP file."""

    template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])

    # Try each ZIP file in TEMPLATE_ZIP_FILES.
    for fname in template_zipfiles:
        try:
            z = zipfile.ZipFile(fname)
            source = z.read(template_name)
        except (IOError, KeyError):
            continue
        z.close()
        # We found a template, so return the source.
        template_path = "%s:%s"% (fname, template_name)
        return (source, template_path)

    # If we reach here, the template couldn't be loaded
    raise TemplateDoesNotExist(template_name)

# This loader is always usable (since zipfile is included with 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 `mysite.zip_loader` entonces agregaremos `mysite.zip_loader.load_templ` a `TEMPLATE_LOADERS`.

10.6. Usando 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 verla, ve a la interfaz de administración y haz click en el 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.

La página *views* es la más valiosa. Cada URL en tu sitio tiene allí una entrada separada. Si la vista relacionada incluye una docstring, haciendo click en la URL te mostrará lo siguiente:

- El nombre de la función de vista que genera esa vista.
- Una breve descripción de 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.

10.7. Configurando el sistema de plantillas en modo autónomo

Nota

Esta 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 un 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 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 descripta 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 `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 descriptas en el Apéndice E y todos las variables cuyos nombres comienzan con `TEMPLATE_` son de obvio interés.

10.8. ¿Qué sigue?

Hasta ahora este libro ha asumido que el contenido que estás visualizando es HTML. Esta no es una suposición incorrecta para un libro sobre desarrollo Web, pero en algunas ocasiones querrás usar Django para generar otros formatos de datos.

El **‘próximo capítulo‘_** describe cómo puedes usar Django para producir imágenes, PDFs y cualquier otro formato de datos que puedas imaginar.

Duplicate explicit target name: “próximo capítulo”.

Capítulo 11

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, PDFs, imágenes, y así sucesivamente.

Hasta ahora nos hemos concentrado en el caso común de la producción de HTML, pero en ese capítulo tomaremos un 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.

11.1. 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* Django *debe*

- Aceptar una instancia `HttpRequest` como primer argumento.
- Retornar una instancia `HttpResponse`.

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 my_image(request):
    image_data = open("/path/to/my/image.png", "rb").read()
    return HttpResponse(image_data, mimetype="image/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.

11.2. Producción de CSV

CSV es formato de datos sencillo que suele ser usada por *software* 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:

```
Year,Unruly Airline Passengers
1995,146
1996,184
1997,235
1998,200
1999,226
2000,251
2001,299
2002,273
2003,281
2004,304
2005,203
```

Nota

El listado precedente contiene números reales; cortesía de la Administración Federal de Aviación (FAA) de E.E.U.U. Vea http://www.faa.gov/data_statistics/passengers_cargo/unruly_passengers/.

Aunque CSV parezca simple, no es un formato que ha sido definido formalmente. Diferentes piezas de software producen y consumen diferentes variantes de CSV, haciendo un poco complicado usarlo. Afortunadamente, Python incluye una biblioteca estándar para CSV, `csv`, que es bastante robusta.

Debido a que el módulo `csv` opera sobre objetos similares a ficheros, es muy fácil usar un `HttpResponse` en lugar de un fichero:

```
import csv
from django.http import HttpResponse

# Número de pasajeros problematicos por año entre 1995 - 2005. En una aplicación real
# esto vendría desde una base de datos o cualquier otro medio de almacenamiento.
UNRULY_PASSENGERS = [146,184,235,200,226,251,299,273,281,304,203]

def unruly_passengers_csv(request):
    # Creamos el objeto Httpresponse con la cabecera CSV apropiada.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=unruly.csv'

    # Creamos un escritor CSV usando a HttpResponse como "fichero"
    writer = csv.writer(response)
    writer.writerow(['Year', 'Unruly Airline Passengers'])
    for (year, num) in zip(range(1995, 2006), UNRULY_PASSENGERS):
        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). Este 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 de tipo `HttpResponse` se ajustan.
- 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 cuántos ejemplos más.

11.3. Generando PDFs

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 PDFs se usan normalmente cuando se necesita entregar un documento a alguien para que lo imprima.

Puedes generar PDFs 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 PDFs dinámicamente es que puedes crear PDFs 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.

11.3.1. Instalando ReportLab

Antes de que puedas generar ningún PDF, deberás 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 sólo disponible en formato PDF) en `http://www.reportlab.org/rsrc/userguide.pdf` contiene instrucciones de instalación adicionales.

> **Nota**
>
> 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.
> Por ejemplo, si estás usando la (excelente) distribución Ubuntu, un simple `apt-get install python-reportlab` hará la magia necesaria.

Prueba tu instalación importando la misma en el intérprete interactivo Python:

```
>>> import reportlab
```

Si ese comando no lanza ningún error, la instalación funcionó.

11.3.2. Escribiendo tu Vista

Del mismo modo que CSV, la generación de PDFs en forma dinámica con Django es sencilla porque la API ReportLab actúa sobre objetos similares a ficheros (*file-like* según la jerga Python).

A continuación un ejemplo "Hola Mundo":

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Se crea el objeto HttpResponse con los headers PDF apropiados.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'
```

```
    # Se crea el objeto PDF, usando el objeto de respuesta como si fuese un "archivo".
    p = canvas.Canvas(response)

    # Se dibujan cosas en el PDF. Aquí es donde se crea el PDF.
    # Lee la documentación de ReportLab para una lista de funcionalidades.
    p.drawString(100, 100, "Hello world.")

    # Se cierra el objeto PDF, y terminamos.
    p.showPage()
    p.save()
    return response
```

Son necesarias alguna notas:

- 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.
- Interactuar con la API ReportLab es sencillo: sólo pasa `response` como el primer argumento a `canvas.Canvas`. La clase `Canvas` espera un objeto *file-like*, y 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).

11.3.3. PDFs complejos

Si estás creando un documento PDF complejo (o cualquier pieza de datos de gran tamaño), considera usar la biblioteca `cStringIO` como un lugar de almacenamiento temporario para tu fichero PDF. La biblioteca `cStringIO` provee una interfaz vía objetos *file-like* que está escrita en C para máxima eficiencia.

Ese es el ejemplo "Hola Mundo" anterior modificado para usar `cStringIO`:

```
from cStringIO import StringIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Se crea el objeto HttpResponse con los headers PDF apropiados.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    temp = StringIO()

    # Se crea el objeto PDF, usando el objeto StringIO como si fuese un "archivo".
    p = canvas.Canvas(temp)

    # Se dibujan cosas en el PDF. Aquí es donde se crea el PDF.
    # Lee la documentación de ReportLab para una lista de funcionalidades.
    p.drawString(100, 100, "Hello world.")

    # Se cierra el objeto PDF.
    p.showPage()
    p.save()

    # Lee el valor del buffer de StringIO y escribe la respuesta.
    response.write(temp.getvalue())
    return response
```

11.4. 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 bonitas e ingeniosas para generar cierto tipo de contenido no-HTML.

11.5. 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.

> **¿Qué es RSS? ¿Qué es Atom?**
>
> 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 debes hacer es escribir una corta 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 (ver los Capítulos 3 y 8 para más información sobre URLconfs).

11.5.1. Inicialización

Para activar los feeds de sindicación en tu sitio Django, agrega lo siguiente en tu URLconf:

```
(r'^feeds/(?P<url>.*)/$',
 'django.contrib.syndication.views.feed',
 {'feed_dict': feeds}
),
```

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 a tus necesidades).

Esta línea de URLconf tiene un argumento extra: `{'feed_dict': feeds}`. Usa este argumento extra para pasar al framework de feeds de sindicación los feeds que deben ser publicados en dicha URL.

Específicamente, `feed_dict` debe ser un diccionario que mapee el *slug* (etiqueta corta de URL) de un feed a la clase Feed. Puedes definir el `feed_dict` en el mismo URLconf. Este es un ejemplo completo de URLconf:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

El ejemplo anterior registra dos feeds:

- El feed representado por `LatestEntries` residirá en `feeds/latest/`.
- El feed representado por `LatestEntriesByCategory` residirá en `feeds/categories/`.

Una vez que este configurado, necesitas definir la propia clase `Feed`.

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).

La clase `Feed` debe se una subclase de `django.contrib.syndication.feeds.Feed`. Esta puede residir en cualquier parte del árbol de código.

11.5.2. Un Feed simple

Este ejemplo simple, tomado de chicagocrime.org, describe un feed que muestra los últimos cinco items agregados:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem

class LatestEntries(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]
```

Las cosas importantes a tener en cuenta son:

- La clase es subclase de `django.contrib.syndication.feeds.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>`. Necesitamos decirle al framework qué datos debe poner en cada uno de los elementos.

- Para especificar el contenido de `<title>` y `<description>`, crea plantillas Django (ver Capítulo 4) llamadas `feeds/latest_title.html` y `feeds/latest_description.html`, donde `latest` es el `slug` especificado en URLconf para el feed dado. Notar que la extensión `.html` es requerida.

  El sistema RSS renderiza 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 `get_absolute_url()` y `item_link()` deben retornar la URL del ítem como una cadena normal de Python.

- Para el ejemplo anterior `LatestEntries`, podemos usar plantillas de feed muy simples. `latest_title.html` contiene:

  ```
  {{ obj.title }}
  ```

  y `latest_description.html` contiene:

  ```
  {{ obj.description }}
  ```

  Es casi demasiado fácil...

11.5.3. Un Feed más complejo

El framework también permite la creación de feeds más complejos mediante el uso de parámetros.

Por ejemplo, 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 en la URL del feed.

En chicagocrime.org, los feed por departamento de policía son accesibles mediante URLs como estas:

- `http://www.chicagocrime.org/rss/beats/0613/`: Retorna los crímenes más recientes para el departamento 0613
- `http://www.chicagocrime.org/rss/beats/1424/`: Retorna los crímenes más recientes para el departamento 1424

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.

un ejemplo aclarará esto. Este es el código para los feeds por departamento:

```
from django.core.exceptions import ObjectDoesNotExist

class BeatFeed(Feed):
    def get_object(self, bits):
        # En caso de "/rss/beats/0613/foo/bar/baz/", u otro
        # revoltijo, corroborar que bits tiene sólo un miembro.
        if len(bits) != 1:
            raise ObjectDoesNotExist
        return Beat.objects.get(beat__exact=bits[0])

    def title(self, obj):
        return "Chicagocrime.org: 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):
        crimes =  Crime.objects.filter(beat__id__exact=obj.id)
        return crimes.order_by('-crime_date')[:30]
```

Aquí tenemos el algoritmo básico del framework RSS, asumiendo esa clase y un requerimiento a la URL `/rss/beats/0613/`:

#. El framework toma la URL `/rss/beats/0613/` y nota que la URL contiene una parte extra tras el slug. Separa esa cadena remanente por el carácter `"/"` y llama al método `get_object()` de la clase `Feed` pasándole los trozos (*bits*) resultantes.

   En este caso, los trozos "son" `['0613']`. Para un requerimiento a `/rss/beats/0613/foo/bar/`, serán `['0613', 'foo', 'bar']`.

1. `get_object()` es el responsable de obtener el departamento requerido, a partir del `bits` dado.
   En este caso, usa la API de base de datos de Django para obtener el departamento. Notar que `get_object()` debe capturar la excepción `django.core.exceptions.ObjectDoesNotExist` si recibe parámetros inválidos. No hay `try/except` abarcando la llamada a `Beat.objects.get()` porque no es necesario. Esa función, ante una falla lanza la excepción `Beat.DoesNotExist`, y `Beat.DoesNotExist` es una subclase de `ObjectDoesNotExist`. Lanzar la excepción `ObjectDoesNotExist` en `get_object()` le dice a Django que produzca un error 404 error para el requerimiento en curso.
2. Para generar los campos `<title>`, `<link>`, y `<description>` del feed, Django usa los métodos `title()`, `link()`, y `description()`. En el ejemplo anterior, se utilizaron atributos simples de clase string, pero este ejemplo muestra que estos pueden ser strings o métodos. Por cada `title`, `link`, y `description`, Django sigue este algoritmo:
   1. 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.
3. Finalmente, nota que `items()` en el ejemplo 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).

La documentación completa de todos los métodos y atributos de las clases `Feed` siempre esta disponible en la documentación oficial de Django (http://www.djangoproject.com/documentation/0.96/syndication_feeds/).

11.5.4. Especificando 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 tu clase `Feed`:

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

Nota que asignas como valor de `feed_type` una clase, no a una instancia. Los tipos de feeds disponibles actualmente se muestran en la Tabla 11-1.

Cuadro 11.1: Tipos de Feeds

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

11.5.5. Enclosures

Para especificar *enclosures* (p. ej. recursos multimedia asociados al ítem del feed tales como feeds de podcasts MP3), usa los ganchos `item_enclosure_url`, `item_enclosure_length`, y `item_enclosure_mime_type`, por ejemplo:

```
from myproject.models import Song

class MyFeedWithEnclosures(Feed):
    title = "Example feed with enclosures"
    link = "/feeds/example-with-enclosures/"

    def items(self):
        return Song.objects.all()[:30]

    def item_enclosure_url(self, item):
        return item.song_url

    def item_enclosure_length(self, item):
        return item.song_length

    item_enclosure_mime_type = "audio/mpeg"
```

Esto asume, por supuesto, que has creado un objeto `Song` con los campos `song_url` y `song_length` (p. ej. el tamaño en bytes).

11.5.6. Idioma

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`.

11.5.7. 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`.

11.5.8. Publicando 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 completo:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
```

Y este es el URLconf asociado:

```
from django.conf.urls.defaults import *
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

feeds = {
    'rss': RssSiteNewsFeed,
    'atom': AtomSiteNewsFeed,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

11.6. 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). 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, vea 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.

11.6.1. Instalación

Para instalar la aplicación sitemap, sigue los siguientes pasos:

1. Agrega `'django.contrib.sitemaps'` a tu variable de configuración `INSTALLED_APPS`.

#. Asegúrate de que `'django.template.loaders.app_directories.load_template_source'` está en tu variable de configuración `TEMPLATE_LOADERS`. Por omisión se encuentra activado, por lo que los cambios son necesarios solamente si modificaste dicha variable de configuración.

1. Asegúrate de que tienes instalado el framework sites (ver Capítulo 14).

> **Nota**
>
> 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 `load_template_source` pueda encontrar las plantillas incluídas.

11.6.2. Inicialización

Para activar la generación del sitemap en tu sitio Django, agrega la siguiente línea a tu URLconf:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
```

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 `news`) a tu clase `Sitemap` (p.e., `BlogSitemap` o `NewsSitemap`). También mapea hacia una *instancia* de una clase `Sitemap` (p. ej. `BlogSitemap(some_var)`).

11.6.3. 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 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 de blog, con un modelo `Entry`, y quieres que tu sitemap incluya todos los enlaces a las entradas individuales de tu Blog. Tu clase `Sitemap` debería verse así:

```
from django.contrib.sitemaps import Sitemap
from mysite.blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

Declarar un `Sitemap` debería verse muy similar a declarar un `Feed`; esto es justamente un objetivo del diseño.

En manera similar a las clases `Feed`, los miembros de `Sitemap` pueden ser métodos o atributos. Ver los pasos en la sección "Un feed más complejo" para más información sobre cómo funciona esto.

Una clase `Sitemap` puede definir los siguientes métodos/atributos:

- `items` (**requerido**): Provee una lista de objetos. Al framework no le importa que *tipo* de objeto es; todo lo que 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. Aquí "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.

11.6.4. 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 ver el Capítulo 14.

Sitemap Genérico

La clase GenericSitemap trabaja con cualquier vista genérica (ver Capítulo 9) que pudieras poseer con anterioridad.

Para usarla, crea una instancia, pasándola en el mismo info_dict que se pasa a la vista genérica. El único requerimiento es que el diccionario tenga una entrada queryset. También debe poseer una entrada 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 los argumentos palabra clave (*keyword*) priority y changefreq al constructor GenericSitemap para especificar dichos atributos para todas las URLs.

Este es un ejemplo de URLconf usando tanto, FlatPageSitemap como GenericSiteMap (con el anterior objeto hipotético Entry):

```
from django.conf.urls.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

sitemaps = {
    'flatpages': FlatPageSitemap,
    'blog': GenericSitemap(info_dict, priority=0.6),
}

urlpatterns = patterns('',
    # some generic view using info_dict
    # ...

    # the sitemap
    (r'^sitemap.xml$',
     'django.contrib.sitemaps.views.sitemap',
     {'sitemaps': sitemaps})
)
```

11.6.5. Creando 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 única diferencias de uso son:

- Usas dos vistas en tu URLconf: `django.contrib.sitemaps.views.index` y `django.contrib.sitemaps.views.si`
- La vista `django.contrib.sitemaps.views.sitemap` debe tomar un argumento de palabra clave llamado `section`.

Así deberían verse las líneas relevantes en tu URLconf para el ejemplo anterior:

```
(r'^sitemap.xml$',
 'django.contrib.sitemaps.views.index',
 {'sitemaps': sitemaps}),

(r'^sitemap-(?P<section>.+).xml$',
 'django.contrib.sitemaps.views.sitemap',
 {'sitemaps': sitemaps})
```

Esto genera automáticamente un fichero `sitemap.xml` que hace referencia a ambos ficheros `sitemap-flatpages.xml` y `sitemap-blog.xml`. La clase `Sitemap` y el diccionario `sitemaps` no cambian en absoluto.

11.6.6. Haciendo ping a Google

Puedes desear hacer un "ping" a Google cuando tu sitemap cambia, para hacerle saber que debe reindexar tu sitio. El framework provee una función para hacer justamente eso: `django.contrib.sitemaps.ping_google()`.

> **Nota**
>
> Hasta el momento en que este libro se escribió, únicamente Google responde a los pings de sitemap. Pero es muy probable que pronto Yahoo y/o MSN también admitan estos pings. Cuando eso suceda, cambiaremos el nombre de `ping_google()` a algo como `ping_search_engines()`, así que asegúrate de verificar la ultima documentación de sitemap en http://www.djangoproject.com/documentation/0.96/sitemaps/.

`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 Entry(models.Model):
    # ...
    def save(self):
        super(Entry, self).save()
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a variety
            # of HTTP-related exceptions.
            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()`.

11.7. ¿Qué sigue?

A continuación, seguiremos indagando más profundamente en las herramientas internas que Django nos ofrece. El Capítulo 12 examina todas las herramientas que necesitas para proveer sitios personalizados: sesiones, usuarios, y autenticación.

¡Adelante!

Capítulo 12

Sesiones, usuario 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.

12.1. 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:T` 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 un rato antes. 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).

12.1.1. 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 como leer y definir *cookies* a bajo nivel. Esto debería ayudarte a entender como 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 show_color(request):
    if "favorite_color" in request.COOKIES:
        return HttpResponse("Your favorite color is%s"% \
            request.COOKIES["favorite_color"])
    else:
        return HttpResponse("You don't have a favorite color.")
```

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* `favorite_color` utilizando el valor que se le pasa como parámetro `GET`:

```
def set_color(request):
    if "favorite_color" in request.GET:

        # Create an HttpResponse object...
        response = HttpResponse("Your favorite color is now%s"% \
            request.GET["favorite_color"])

        # ... and set a cookie on the response
        response.set_cookie("favorite_color",
                            request.GET["favorite_color"])

        return response

    else:
        return HttpResponse("You didn't give a favorite color.")
```

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 12-1.

Cuadro 12.1: Opciones de las Cookies

Parámetro	Valor por omisión	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.

Cuadro 12.1: Opciones de las Cookies

Parámetro	Valor por omisión	Descripción
`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).

12.1.2. 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 cualesquiera razones.

- 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. El **'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 cuantos sitios web cometen este tipo de error; no lleva más de unos segundos engañar a sus sistemas de "seguridad".

12.2. 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 por ti.

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 aisla de la mayoría de los problemas asociados con las cookies.

Veamos como activar las sesiones, y como usarlas en nuestras vistas.

12.2.1. Activar sesiones

Las sesiones se implementan mediante un poco de *middleware* (véase Capítulo 15) 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.SessionMiddl`
2. Comprobar que `'django.contrib.sessions'` esté incluido en el valor de `INSTALLED_APPS` (Y ejecutar `manage.py syncdb` si lo has tenido que añadir).

Los valores por defecto creados por `startproject` ya tienes estas dos características habilitadas, 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 borrar `'django.contrib.sessions'` de `INSTALLED_APPS`. Esto te permitirá disminuir mínimamente la sobrecarga del servidor, pero toda ayuda es buena.

12.2.2. Usar las sesiones en una vista

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:

```
# Set a session value:
request.session["fav_color"] = "blue"

# Get a session value -- this could be called in a different view,
# or many requests later (or both):
fav_color = request.session["fav_color"]

# Clear an item from the session:
del request.session["fav_color"]

# Check if the session has a given key:
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 reservadas 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 `has_commented` 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 post_comment(request, new_comment):
    if request.session.get('has_commented', False):
```

```
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

Esta vista simplificada permite que un usuario se identifique como tal en nuestras páginas:

```
def login(request):
    try:
        m = Member.objects.get(username__exact=request.POST['username'])
        if m.password == request.POST['password']:
            request.session['member_id'] = m.id
            return HttpResponse("You're logged in.")
    except Member.DoesNotExist:
        return HttpResponse("Your username and password didn't match.")
```

Y esta le permite cerrar o salir de la sesión:

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

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 ejemplo son deliberadamente simples para que se comprendan con más facilidad.

12.2.3. Comprobar que las *cookies* sean utilizables

Como ya mencionamos, no se puede confiar en que el 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):

    # If we submitted the form...
    if request.method == 'POST':

        # Check that the test cookie worked (we set it below):
        if request.session.test_cookie_worked():

            # The test cookie worked, so delete it.
            request.session.delete_test_cookie()

            # In practice, we'd need some logic to check username/password
            # here, but since this is an example...
            return HttpResponse("You're logged in.")

        # The test cookie failed, so display an error message. If this
```

```
        # was a real site we'd want to display a friendlier message.
        else:
            return HttpResponse("Please enable cookies and try again.")

    # If we didn't post, send the test cookie along with the login form.
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

> **Nota**
>
> De nuevo, las funciones de autentificación ya definidas en el entorno realizan estos chequeos por ti.

12.2.4. Usar las 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, 12)
```

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}
```

12.2.5. Cuándo se salvan las sesiones

Django, en principio, solo salva la sesión en la base de datos si esta 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:

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of 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 salvará 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.

12.2.6. 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 del 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.

12.2.7. 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 12-2.

Cuadro 12.2: Valores de configuración que influyen en el comportamiento de las cookies

Opción	Descripción	Valor por defecto
`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`

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 `pickled`. 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.

12.3. 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 dicen 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 llamada 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:

- *Usuarios*: Personas registradas en tu sitio web
- *Permisos*: Valores binarios (Si/No) que indican si un usuario puede o no realizar una tarea determinada.
- *grupos*: Una forma genérica de aplicar etiquetas y permisos a más de un usuario.
- *mensajes*: Un mecanismo sencillo que permite enviar y mostrar mensajes del sistema usando una cola.
- *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.

12.3.1. Habilitando 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 usuario implica usar cookies, y por lo tanto necesitamos el entorno de sesiones operativo.
- Incluye `'django.contrib.auth'` dentro de tu `INSTALLED_APPS` y ejecuta `manage.py syncdb`.
- 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():
    # Do something for authenticated users.
else:
    # Do something for anonymous users.
```

12.4. Utilizando usuarios

Una vez que ya tienes un usuario (normalmente mediante `request.user`, pero 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 12-3 y 12-4 listan todos los campos y métodos, respectivamente, de los objetos de la clase `User`.

Cuadro 12.3: 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 "**'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.

Cuadro 12.4: 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* salva 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.
`get_profile()`	Devuelve un perfil del usuario específico para el sitio. En la sección “Perfiles” se explicará con más detalle este método.

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:

```
# Set a user's groups:
myuser.groups = group_list

# Add a user to some groups:
myuser.groups.add(group1, group2,...)

# Remove a user from some groups:
myuser.groups.remove(group1, group2,...)
```

```
# Remove a user from all groups:
myuser.groups.clear()

# Permissions work the same way
myuser.permissions = permission_list
myuser.permissions.add(permission1, permission2, ...)
myuser.permissions.remove(permission1, permission2, ...)
myuser.permissions.clear()
```

12.4.1. 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 como hacer que los usuario pueden 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 "Correct!"
... else:
...     print "Oops, that's wrong!"
```

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 login(request):
    username = request.POST['username']
    password = request.POST['password']
    user = auth.authenticate(username=username, password=password)
    if user is not None and user.is_active:
        # Correct password, and the user is marked "active"
        auth.login(request, user)
        # Redirect to a success page.
        return HttpResponseRedirect("/account/loggedin/")
    else:
        # Show an error page
        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)
    # Redirect to a success page.
    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 = patterns('',
    # existing patterns here...
    (r'^accounts/login/$',  login),
    (r'^accounts/logout/$', 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">Sorry, that's not a valid username or password</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 action='.' method='post'>

{% 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.

12.4.2. 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_authenti` y redirigir a una página de identificación, si procede:

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/login/?next=%s'% request.path)
    # ...
```

O quizás mostrar un mensaje de error:

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('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 my_view(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 esta identificado correctamente

12.4.3. 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 `polls.can_vote` (Se explicará esto de los permisos con más detalle dentro de poco) haríamos:

```
def vote(request):
    if request.user.is_authenticated() and request.user.has_perm('polls.can_vote')):
        # vote here
    else:
        return HttpResponse("You can't vote in this poll.")
```

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 user_can_vote(user):
    return user.is_authenticated() and user.has_perm("polls.can_vote")

@user_passes_test(user_can_vote, login_url="/login/")
def vote(request):
    # Code here can assume a logged-in user with the correct permission.
    ...
```

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('polls.can_vote', login_url="/login/")
def vote(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, y apuntar en tu URLconf al recubrimiento en vez de a la vista genérica:

```
from dango.contrib.auth.decorators import login_required
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

Puedes cambiar el decorador `login_required` por cualquier otro que quieras usar, como es lógico.

12.4.4. Gestionar usuarios, permisos y grupos

La forma más fácil de gestionar el sistema de autentificación es a través de la interfaz de administración `admin`. El Capítulo 6 describe como 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 momento, `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 salvarlos, si quieres:

```
>>> user.is_staff = True
>>> user.save()
```

Cambia contraseñas

Puedes cambiar las contraseña de un usuario llamando a `set_password()`:

```
>>> user = User.objects.get(username='john')
>>> user.set_password('goo goo goo joob')
>>> 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 si por el carácter dolar ($).

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 del usuario

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 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:

```
from django import oldforms as forms
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.contrib.auth.forms import UserCreationForm

def register(request):
    form = UserCreationForm()

    if request.method == 'POST':
        data = request.POST.copy()
        errors = form.get_validation_errors(data)
        if not errors:
            new_user = form.save(data)
            return HttpResponseRedirect("/books/")
    else:
        data, errors = {}, {}

    return render_to_response("registration/register.html", {
        'form' : forms.FormWrapper(form, data, errors)
    })
```

Este formulario asume que existe una plantilla llamada `registration/register.html`. esa plantilla podría consistir en algo parecido a esto:

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

{% block title%}Create an account{% endblock%}

{% block content%}
  <h1>Create an account</h1>
  <form action="." method="post">
    {% if form.error_dict%}
      <p class="error">Please correct the errors below.</p>
    {% endif%}

    {% if form.username.errors%}
      {{ form.username.html_error_list }}
    {% endif%}
    <label for="id_username">Username:</label> {{ form.username }}

    {% if form.password1.errors%}
      {{ form.password1.html_error_list }}
    {% endif%}
    <label for="id_password1">Password: {{ form.password1 }}

    {% if form.password2.errors%}
      {{ form.password2.html_error_list }}
    {% endif%}
    <label for="id_password2">Password (again): {{ form.password2 }}

    <input type="submit" value="Create the account" />
  </label>
{% endblock%}
```

> **Nota**
>
> `django.contrib.auth.forms.UserCreationForm` es, a la hora de publicar esto, un formulario del estilo *oldforms*. Véase http://www.djangoproject.com/documentation/0.96/forms/ para más detalles sobre *oldforms*. La transición al nuevo sistema, tal y como se explica en el capítulo siete, será completada muy pronto.

12.4.5. 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 el **'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>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else%}
  <p>Welcome, new user. Please log in.</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.polls%}
  <p>You have permission to do something in the polls app.</p>
  {% if perms.polls.can_vote%}
    <p>You can vote!</p>
  {% endif%}
{% else%}
  <p>You don't have permission to do anything in the polls app.</p>
{% endif%}
```

12.5. El resto de detalles: permisos, grupos, mensajes y perfiles

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.

12.5.1. 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 añada a la tabla `auth_permission` cuando ejecutas `manage.py syncdb`.

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"`.

Hay que tener cuidado de que el modelo tenga creada una clase `Admin` a la hora de ejecutar `syncdb`. Si no la tiene, no se crearán los permisos. Si has inicializado la base de datos y has añadido la clase `Admin` con posterioridad, debes ejecutar otra vez `syncdb` para crear los permisos.

También puedes definir tus propios permisos para un modelo, usando el atributo `permissions` en la clase `Meta`. El siguiente ejemplo crea tres permisos hechos a medida:

```
class USCitizen(models.Model):
    # ...
    class Meta:
        permissions = (
            # Permission identifier     human-readable permission name
            ("can_drive",               "Can drive"),
            ("can_vote",                "Can vote in elections"),
            ("can_drink",               "Can drink alcohol"),
        )
```

Esto permisos sólo se crearán cuando ejecutes `syncdb`. Es responsabilidad tuya comprobar estos permisos en tus vistas.

Igual que con los usuarios, los permisos se implementa en un modelo Django que reside en el módulo `django.contrib.auth.model` Esto significa que puedes usar la API de acceso a la base de datos para interactuar con los permisos de la forma que quieras.

12.5.2. 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.

12.5.3. Mensajes

El sistema de mensajes es un 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 salva un mensaje para el usuario después de crear una lista de reproducción:

```
def create_playlist(request, songs):
    # Create the playlist with the given songs.
    # ...
    request.user.message_set.create(
        message="Your playlist was added successfully."
    )
    return render_to_response("playlists/create.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.

12.5.4. Perfiles

La parte final de nuestro puzzle consiste en el sistema de perfiles. Para entender mejor que es este sistema y para que sirve, veamos primero el problema que se supone tiene que resolver.

El resumen sería este: Muchos sitios en Internet necesitan almacenar más información acerca de sus usuarios de la que está disponible en un objeto de la clase `User`. Para resolver este problema, otros entornos definen una serie de campos "extra". Django, por el contrario, ha optado por proporcionar un mecanismo sencillo que permita crear un perfil con los datos que queramos, y que queda enlazado con la cuenta de usuario. Estos perfiles pueden incluso ser diferentes según cada proyecto, y también se pueden gestionar diferentes perfiles para sitios diferentes usando la misma base de datos.

El primer paso para crear un perfil es definir el modelo que va a almacenar la información que queremos guardar. El único requerimiento que Django impone a este modelo es que disponga de una campo de tipo `ForeignKey`, que sea único (`unique=True`) y que lo vincule con el modelo `User`. Además, el campo debe llamarse `user`. Aparte de eso, puedes usar tantos y tan variados campos de datos como quieres. El siguiente ejemplo muestra un perfil absolutamente arbitrario:

```
from django.db import models
from django.contrib.auth.models import User

class MySiteProfile(models.Model):
    # This is the only required field
    user = models.ForeignKey(User, unique=True)

    # The rest is completely up to you...
    favorite_band = models.CharField(maxlength=100, blank=True)
    favorite_cheese = models.CharField(maxlength=100, blank=True)
    lucky_number = models.IntegerField()
```

El siguiente paso es decirle a Django donde buscar estos perfiles. Para ello asigna a la variable de configuración `AUTH_PROFILE_MODULE` el identificador de tu modelo. Así, si el perfil que definimos en el ejemplo anterior residiera en una aplicación llamada "myapp", pondrías esto en tu fichero de configuración:

```
AUTH_PROFILE_MODULE = "myapp.mysiteprofile"
```

Una vez hecho esto, se puede acceder al perfil del usuario llamando a `user.get_profile()`. Esta función elevará una excepción de tipo `DoesNotExist` si el usuario no tiene creado el perfil (puedes atrapar esta excepción y crear el perfil en ese momento).

12.6. ¿Qué sigue?

Si, la verdad es que el sistema de autorización tiene tela que cortar. La mayor parte 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 todas las utilidades posibles a mano.

En el **'próximo capítulo'_**, echaremos un vistazo a una parte de Django que necesita la infraestructura que proporciona el sistema de usuarios/sesiones de Django: la aplicación de comentarios. Esta aplicación permite añadir, de forma muy sencilla, un completo sistema de comentarios -por parte de usuarios anónimos o identificados- a cualquier tipo de objeto que queramos. ¡Hasta el infinito y más allá!

Duplicate explicit target name: "próximo capítulo".

Capítulo 13

Cache

Los sitios Web estáticos, en las que las páginas son servidas directamente a la 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.com o Slashdot; 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 como 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 "upstream", tales como Squid (http://www.squid-cache.org/) 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 como usar el sistema de cache de Django. Cuando tu sitio se parezca cada vez más a Slashdot, estarás contento de entender este material.

13.1. Activando el 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 tu 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 `CACHE_BACKEND` en tu archivo de configuración. Si usas cache y no especificas `CACHE_BACKEND`, Django usará `simple:///` por omisión. Las siguientes secciones explican todos los valores disponibles para `CACHE_BACKEND`.

13.1.1. Memcached

Por lejos la más rápida, el tipo de cache más eficiente para Django, Memcached es un framework de cache 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 libremente disponible en http://danga.com/memcached/. Corre como un demonio y se le asigna una cantidad específica de memoria RAM. Su característica principal es proveer una interfaz--una *super-liviana-y-rápida* interfaz--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 los *bindings* Python para Memcached, los cuales no vienen con Django. Dichos *bindings* vienen en un módulo de Python, `memcache.py`, el cual está disponible en http://www.tummy.com/Community/software/python-memcached/.

Para usar Memcached con Django, coloca `CACHE_BACKEND` como `memcached://ip:puerto/`, donde `ip` es la dirección IP del demonio de Memcached y `puerto` es el puerto donde Memcached está corriendo.

En el siguiente ejemplo, Memcached está corriendo en localhost (127.0.0.1) en el puerto 11211:

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

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 `CACHE_BACKEND`, 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:

```
CACHE_BACKEND = 'memcached://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):

```
CACHE_BACKEND = 'memcached://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 particularmente temporaria.

13.1.2. 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, coloca la propiedad `CACHE_BACKEND` como `"db://nombre_tabla"`, donde `nombre_tabla` es el nombre de la tabla en la base de datos. En el siguiente ejemplo, el nombre de la tabla para el cache es `mi_tabla_cache`:

```
CACHE_BACKEND = 'db://mi_tabla_cache'
```

El sistema de cache usará la misma base de datos especificada en el archivo de configuración. No podrás usar un base de datos diferente para tal.

13.1.3. Cache en Sistema de Archivos

Para almacenar la cache en el sistema de archivos, coloca el tipo `"file://"` en la propiedad `CACHE_BACKEND`, especificando 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:

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

Observa que hay tres barras invertidas en el comienzo del ejemplo anterior. Las primeras dos son para `file://`, y la tercera es el primer caracter de la ruta del directorio, `/var/tmp/django_cache`. Si estás en Windows, coloca la letra correspondiente al disco después de `file://`, como aquí:: `file://c:/foo/bar`.

La ruta del directorio 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 de sistema uado 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` 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.

13.1.4. Cache en Memoria local

Si quieres 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 thread-safe, pero no es tan eficiente como Memcache dada su estrategia de bloqueo simple y reserva de memoria.

Para usarla, coloca `CACHE_BACKEND` como `'locmem:///'`, por ejemplo:

```
CACHE_BACKEND = 'locmem:///'
```

13.1.5. Cache Simple (para desarrollo)

Una cache simple, y de un solo proceso en memoria, está disponible como `'simple:///'`, por ejemplo:

```
CACHE_BACKEND = 'simple:///'
```

Esta cache apenas guarda los datos en proceso, lo que significa que sólo debe ser usada para desarrollo o testing.

13.1.6. Cache Dummy (o estúpida)

Finalmente, Django incluye una cache "dummy" que no realiza cache; sólo implementa la interfaz de cache sin realizar ninguna acción.

Esto es útil cuando tienes un sitio en producción que usa mucho cache en varias partes y en un entorno de desarrollo/prueba en cual no quieres hacer cache. En ese caso, usa `CACHE_BACKEND` como `'dummy:///'` en el archivo de configuración para tu entorno de desarrollo, por ejemplo:

```
CACHE_BACKEND = 'dummy:///'
```

Como resultado de esto, tu entorno de desarrollo no usará cache, pero tu entorno de producción si lo hará.

13.1.7. Argumentos de CACHE_BACKEND

Cada tipo de cache puede recibir argumentos. Estos son dados como una query-string en la propiedad `CACHE_BACKEND`. Los argumentos válidos son:

- `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.
- `max_entries`: Para la cache simple, 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 actual 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`.

  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 *mucho* más rápido pero al costo de perder más datos de la cache. Este argumento tomará un valor de 3 si no se especifica.

En este ejemplo, `timeout` se fija en `60`:

```
CACHE_BACKEND = "locmem:///?timeout=60"
```

En este ejemplo, `timeout` se fija en `30` y `max_entries` en `400`:

```
CACHE_BACKEND = "locmem:///?timeout=30&max_entries=400"
```

Tanto los argumentos desconocidos asi como los valores inválidos de argumentos conocidos son ignorados silenciosamente.

13.2. La cache por sitio

Una vez que hayas especificado `CACHE_BACKEND`, 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 solamente agrega `'django.middleware.cache.CacheMiddleware'` a la propiedad `MIDDLEWARE_CLASSES`, como en el siguiente ejemplo:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.CacheMiddleware',
    'django.middleware.common.CommonMiddleware',
)
```

> **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_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.

Esta cache middleware admite otras característica, `CACHE_MIDDLEWARE_ANONYMOUS_ONLY`. Si defines esta característica, y la defines como `True`, la cache middleware sólo colocará en cache pedidos anónimos (p.e.: pedidos hechos por un usuario no logueado). Esta es una manera simple y efectiva de deshabilitar la cache para cualquier página de algún usuario específico, como la interfaz de administración de Django. Ten en cuenta que si usas `CACHE_MIDDLEWARE_ANONYMOUS_ONLY`, deberás asegurarte que has activado `AuthenticationMiddleware` y que `AuthenticationMiddleware` aparezca antes de `CacheMiddleware` en tus `MIDDLEWARE_CLASSES`

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`.

13.3. Cache por vista

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 tu 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 my_view(request, param):
    # ...
my_view = cache_page(my_view, 60 * 15)
```

De otra manera, si estás usando la versión 2.4 o superior de Python, puedes usar la sintaxis de un decorador. El siguiente ejemplo es equivalente al anterior:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request, param):
    # ...
```

`cache_page` recibe un único argumento: el tiempo de vida en segundos de la cache. En el ejemplo anterior, el resultado de `my_view()` 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 `my_view`, si tu URLconf se ve como:

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

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.

13.3.1. Especificando la cache por vista en URLconf

Los ejemplos en la sección anterior tienen codificado [18] que la vista se coloque en cache, porque `cache_page` modifica la función `my_view` ahí mismo. Este enfoque acopla tu vista con el sistema de cache, lo cual no es lo ideal por varias razones. Por ejemplo, puede que quieras reusar 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 URLconf. Aquí el URLconf como estaba antes:

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

Ahora la misma cosa con `my_view` envuelto con `cache_page`:

```
from django.views.decorators.cache import cache_page

urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', cache_page(my_view, 60 * 15)),
)
```

Si tomas este enfoque no olvides de importar `cache_page` dentro de tu URLconf.

13.4. 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). N.T.: pickling

Aquí vemos como importar la API:

```
>>> from django.core.cache import cache
```

La interfaz básica es `set(key, value, timeout_seconds)` y `get(key)`:

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

El argumento `timeout_seconds` es opcional y obtiene el valor del argumento `timeout` de `CACHE_BACKEND`, 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`:

```
# Wait 30 seconds for 'my_key' to expire...

>>> cache.get('my_key')
None

>>> cache.get('some_unset_key')
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('my_key', 'has expired')
'has expired'
```

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 key 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'])
{'a': 1, 'b': 2, 'c': 3}
```

Si una key 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 keys 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.

13.5. Caches upstream

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 *upstream*. 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 upstream:

- 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 upstream 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 upstream 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.

13.5.1. Usando 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

# Python 2.3 syntax.
def my_view(request):
    # ...
my_view = vary_on_headers(my_view, 'User-Agent')

# Python 2.4+ decorator syntax.
@vary_on_headers('User-Agent')
def my_view(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 my_view(request):
    # ...
```

Esto le dice a la cache de upstream que diferencie *ambos*, lo que significa que cada combinación de una cookie y un navegador obtendrá 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 my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(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 my_view(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.

13.5.2. Otros Encabezados de cache

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 my_view(request):
    # ...
```

Este decorador se encarga de enviar los encabezados HTTP apropiados detrás de escena.

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 unicamente 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.)

En 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 3.600 segundos:

```
from django.views.decorators.cache import cache_control
@cache_control(must_revalidate=True, max_age=3600)
def my_view(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, lea 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.

13.6. 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.

13.7. 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`

13.8. ¿Qué sigue?

Django incluye un número de paquetes opcionales. Hemos cubierto algunos de los mismos: el sistema de administración (Capítulo 6) y el marco de sesiones/usuarios (Capítulo 11).

El **‘próximo capítulo‘_** cubre el resto de los marcos de trabajos “de la comunidad”. Existen una cantidad interesante de herramientas disponibles; no querrás perderte ninguna de ellas.

Duplicate explicit target name: “próximo capítulo”.

[18] N. del T.: hard-coded

Capítulo 14

Otros sub-frameworks contribuidos

Una de las varias fortalezas de Python, es su filosofía de "baterías incluidas". Cuando instalas Python, viene con una amplia biblioteca de paquetes que puedes comenzar a usar inmediatamente, sin necesidad de descargar nada más. Django trata de seguir esta filosofía, e incluye su propia biblioteca estándar de agregados útiles para las tareas comunes del desarrollo web. Este capítulo cubre dicha colección de agregados.

14.1. La biblioteca estándar de Django

La biblioteca estándar de Django vive en el paquete `django.contrib`. Dentro de cada sub-paquete hay una pieza aislada de funcionalidad para agregar. Estas piezas no están necesariamente relacionadas, pero algunos sub-paquetes de `django.contrib` pueden requerir a otros.

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), pero otros consisten solamente de *middleware* o de etiquetas de plantillas (*template tags*).

La única característica común a 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 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 los capítulos 6 y 18.
- `auth`: el *framework* de autenticación de Django. Consulta el Capítulo 12.
- `comments`: una aplicación para comentarios. Esta aplicación está actualmente bajo un fuerte desarrollo, y por lo tanto, no puede ser cubierta por completo para cuando se publique de este libro. Chequea el sitio web de Django para obtener la última información sobre esta aplicación.
- `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 "Haciendo los datos más humanos" más adelante.
- `markup`: un conjunto de filtros de plantillas de Django, que implementan varios lenguajes de marcado conocidos. Consulta la sección titulada "Filtros de marcado" más adelante.
- `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 el Capítulo 11.
- `sites`: un *framework* que te permite operar múltiples sitios web desde la misma base de datos, y con una única instalación de Django. Consulta la próxima sección, "Sites".
- `syndication`: un *framework* para generar documentos de sindicación (*feeds*), en RSS y en Atom. Consulta el Capítulo 11.

El resto de este capítulo entra en los detalles de cada paquete `django.contrib` que no ha sido cubierto aún en este libro.

14.2. Sites

El sistema *sites* 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.

14.2.1. Escenario 1: reuso de los datos en múltiples sitios

Como explicamos en el Capítulo 1, 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 cabeza dura 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".

14.2.2. 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` (ej. `'LJWorld.com'`) y `domain` (ej. `'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.

14.2.3. Modo de uso del *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 `manage.py syncdb` 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.

14.2.4. Las capacidades del framework Sites

Las siguientes secciones describen las cosas que puedes hacer con este *framework*.

Reuso de los datos en múltiples sitios

Para reusar 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 Article(models.Model):
    headline = models.CharField(maxlength=200)
    # ...
    sites = models.ManyToManyField(Site)
```

Esa es toda la infraestructura necesaria para asociar artículos con múltiples sitios en tu base de datos. Con eso en su lugar, puedes reusar el mismo código de vista para múltiples sitios. Continuando con el modelo `Article` del ejemplo, aquí mostramos cómo luciría una vista `article_detail`:

```
from django.conf import settings

def article_detail(request, article_id):
    try:
        a = Article.objects.get(id=article_id, sites__id=settings.SITE_ID)
    except Article.DoesNotExist:
        raise Http404
    # ...
```

esta función de vista es reusable porque chequea 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 Article(models.Model):
    headline = models.CharField(maxlength=200)
    # ...
    site = 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 my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else.
```

Por supuesto, es horrible meter en el código el ID del sitio de esa manera. Una forma levemente más limpia de lograr lo mismo, es chequear el dominio actual del sitio:

```
from django.conf import settings
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

Este fragmento de código usado para obtener el objeto `Site` según el valor de `settings.SITE_ID` es tan usado, que el administrador 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 my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

Nota

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 register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' % current_site.name,
        'Thanks for your subscription. We appreciate it.\n\n-The %s team.' % current_site.name,
        'editor@%s' % current_site.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 register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'do-not-reply@example.com', [user_email])
    # ...
```

En este caso, debes crear las plantillas `subject.txt` y `message.txt` en ambos directorios de plantillas, el de LJWorld.com y el d Lawrence.com . Como mencionamos anteriormente, eso te da más flexibilidad, pero también es más complejo.

Una buena idea ex explotar los objetos `Site` lo más posible, para que no hayan una complejidad y una redundancia innecesarias.

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 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
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://example.com/mymodel/objects/3/'
```

14.2.5. CurrentSiteManager

Si los `Site``s juegan roles importante en tu aplicación, considera el uso del útil ``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 Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(maxlength=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

Con este modelo, `Photo.objects.all()` retorna todos los objetos `Photo` de la base de datos, pero `Photo.on_site.all()` retorna sólo los objetos `Photo` asociados con el sitio actual, de acuerdo a la opción de configuración `SITE_ID`.

En otras palabras, estas dos sentencias son equivalentes:

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

¿Cómo supo `CurrentSiteManager` cuál campo de `Photo` 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, que tiene un campo llamado `publish_on`, lo demuestra:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(maxlength=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

Si intentas usar `CurrentSiteManager` y pasarle un nombre de campo que no existe, Django lanzará un `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`.

14.2.6. El uso que hace Django del *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 comments*, cada comentario está asociado con un sitio en particular. Cuando un comentario es posteado, su `site` es asignado al `SITE_ID` actual, y cuando los comentarios son listados con la etiqueta de plantillas apropiada, sólo los comentarios del sitio actual son mostrados.
- 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 11), 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` dede el objeto `Site` actual si no especificas un *fully qualified domain*.
- En el *framework authentication* (consulta el Capítulo 12), la vista `django.contrib.auth.views.login` le pasa el nombre del `Site` actual a la plantilla como `{{ site_name }}`.

14.3. Flatpages

A menudo tendrás una aplicación Web impulsada por 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 ese 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").

14.3.1. Usando flatpages

Para instalar la aplicación flatpages, sigue estos pasos:

1. Agrega `'django.contrib.flatpages'` a tu `INSTALLED_APPS`. `django.contrib.flatpages` dependes de `django.contrib.sites`, así 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 `manage.py syncdb` para instalar las dos tables necesarias en tu base de datos.

La aplicación flatpages crea dos tablas en tu base de datos: `django_flatpage` y `django_flatpage_sites`. `django_flatpag` simplemente mantiene una correspondencia entre URLs y títulos más contenido 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 el 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` en o cerca del final de la lista, debido a que se trata de una opción de último recurso.

14.3.2. Agregando, modificando y eliminando flatpages

Puedes agregar, cambiar y eliminar flatpages de dos maneras:

Vía 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.

Vía la API Python

Como ya se describió, las flatpages se representan mediante un modelo Django estándar que reside en `django/contrib/flatpages` 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='/about/',
...     title='About',
...     content='<p>About this site...</p>',
...     enable_comments=False,
...     template_name='',
...     registration_required=False,
... )
>>> fp.save()
>>> fp.sites.add(Site.objects.get(id=1))
>>> FlatPage.objects.get(url='/about/')
<FlatPage: /about/ -- About>
```

14.3.3. Usando 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 la 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>
```

14.4. 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 `/music/` a `/sections/arts/music/`.". 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.

14.4.1. Usando el framework redirects

Para instalar la aplicación redirects, sigue estos pasos:

1. Agrega `'django.contrib.redirects'` a tu `INSTALLED_APPS`.
2. Agrega `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` a tu variable de configuración `MIDDLEWARE_CLASSES`.
3. Ejecuta el comando `manage.py syncdb` para instalar la única tabla necesaria a tu base de datos.

`manage.py syncdb` 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 "Agregando, modificando y eliminando 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.

Notar 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.

> **Nota**
>
> 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 tu podrías decidir lo contrario.

14.4.2. Agregando, modificando y eliminando redirecciones

Puedes agregar, modificar y eliminar redirecciones de dos maneras:

Vía 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.

Vía 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='/music/',
...     new_path='/sections/arts/music/',
... )
>>> red.save()
>>> Redirect.objects.get(old_path='/music/')
<Redirect: /music/ ---> /sections/arts/music/>
```

14.5. 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.

14.5.1. Un ejemplo simple de CSRF

Supongamos que posees una cuenta de *webmail* en `example.com`. Este sitio proveedor de *webmail* tiene un botón *Log Out* que apunta a la URL `example.com/logout` -- esto es, la única acción que necesitas realizar para desconectarte (*log out*) es visitar la página `example.com/logout`.

Un sitio malicioso puede coercerte 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 *webmail* 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 *webmail* 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.

14.5.2. 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.

14.5.3. Previniendo la 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 que lo hace la capa de prevención de CSRF de Django, tal como se explica en la siguiente sección.

Usando 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.

Para activar esta proteccion, 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 desde 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. Revisa la sección "Orden de MIDDLEWARE_CLASSES" en el Capítulo 13 si necesitas conocer más sobre el tema.

En el caso en el que estés interesado, así es como trabaja `CsrfMiddleware`. Realiza las siguientes 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 12 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 estár salteandote 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`.

14.6. Haciendo los datos más humanos

Esta aplicación 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, usa `{% load humanize %}` en una plantilla, y tendrás acceso a los filtros que se describen en las siguientes secciones.

14.6.1. 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".
- 10 se convierte en "10".

Puedes pasarle ya sea un entero o una representación en cadena de un entero.

14.6.2. 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.

14.6.3. 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".
- 1200000000 se convierte en "1.2 millardos".

Se admiten valores hasta un billardo (1,000,000,000,000,000).
Puedes pasarle ya sea un entero o una representación en cadena de un entero.

14.6.4. 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.

14.7. Filtros de marcado

La siguiente colección de filtros de plantilla implementa lenguajes comunes de marcado:

- `textile`: Implementa Textile (http://en.wikipedia.org/wiki/Textile_%28markup_language%29)
- `markdown`: Implementa Markdown (`http://en.wikipedia.org/wiki/Markdown`)
- `restructuredtext`: Implementa ReStructured Text (`http://en.wikipedia.org/wiki/ReStructuredText`)

En cada caso el filtro espera el texto con formato de marcado como una cadena y retorna una cadena representando el texto con formato. Por ejemplo el filtro `textile` convierte texto marcado con formato Textile a HTML:

```
{% load markup %}
{{ object.content|textile }}
```

Para activar estos filtros, agrega `django.contrib.markup` a tu variable de configuración `INSTALLED_APPS`. Una vez que hayas hecho esto, usa `{% load markup %}` en una plantilla y tendrás acceso a dichos filtros. Para más detalles examina el código fuente en `django/contrib/markup/templatetags/markup.py`.

14.8. ¿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 el tuyo propio.

Capítulo 15

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 estas 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 12 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 13 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 contribuidas `flatpages`, `redirects`, y `csrf` del Capítulo 14 hacen su magia a través de componentes middleware.

Este capítulo se sumerge más profundamente en qué es exactamente el middleware y cómo funciona, y explica cómo puedes escribir tu propio middleware.

15.1. Qué es 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 el Capítulo 20). 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 can be a comma-separated list of IPs.
            # Take just the first one.
            real_ip = real_ip.split(",")[0]
            request.META['REMOTE_ADDR'] = real_ip
```

Si esto es instalado (mira la siguiente sección), el valor de `X-Forwarded-For` de todas las peticiones 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. Esta ubicada en `django.middleware.http`, y puedes leer más sobre ella en la siguiente sección.

15.2. 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 un componente middleware, agregarlo a la tupla `MIDDLEWARE_CLASSES` en tu archivo de configuración. En `MIDDLEWARE_CLASSES`, cada componente middleware se representa con un string: 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.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.middleware.doc.XViewMiddleware'
)
```

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 sección "Cómo procesa una petición Django: Detalles completos" en el Capítulo 3 para un repaso de las fases.

15.3. 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.

15.3.1. Inicializar: __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. Tu 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`.

15.3.2. 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. 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`.

15.3.3. 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.

Cuadro 15.1: Argumentos que se pasan a process_view()

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.

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`.

15.3.4. 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.

15.3.5. Pos-procesador de excepción: process_exception(self, request, exception)

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 `exception`, 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

15.4. Middleware incluido

Django viene con algunos middleware incorporados para lidiar con problemas comunes, los cuales discutiremos en las secciones que siguen.

15.4.1. 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 el Capítulo 12 para los detalles completos.

15.4.2. Middleware "Common"

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 esta 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 indizador 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* son 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 que existe un middleware de `GET` condicional, que veremos en breve, el cual maneja ETags y hace algo más.

15.4.3. 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.

15.4.4. 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.

15.4.5. 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['RE` basándose en el valor de `request.META['HTTP_X_FORWARDED_FOR']`, si este último esta 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`.

> **Atención!**
>
> Este middleware *no* hace validar `HTTP_X_FORWARDED_FOR`.
> Si no estas 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`.

15.4.6. Middleware de soporte para sesiones

Clase middleware: `django.contrib.sessions.middleware.SessionMiddleware`.

Este middleware habilita el soporte para sesiones. Mira el Capítulo 12 para más detalles.

15.4.7. 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 13.

15.4.8. 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 C para obtener más información sobre las transacciones de base de datos.

15.4.9. 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.

15.5. ¿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 integrarse con sistemas existentes, tales como esquemas de bases de datos que has heredado de la década de los 80.

Duplicate explicit target name: "próximo capítulo".

Capítulo 16

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 de verde pasto fresco. 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 [19]. Este capítulo explica algunas de las estrategias de integración.

16.1. 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`.

16.1.1. Empleo de `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 esta instalado y tienes una base de datos existente.

1. Crea un proyecto Django ejecutando `django-admin.py startproject mysite` (donde `mysite` es el nombre de tu proyecto). Usaremos `mysite` como nombre de proyecto en este ejemplo.
2. Edita el archivo de configuración en ese proyecto, `mysite/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. Mira el Capítulo 5 para más información).
3. Crea una aplicación dentro de tu proyecto ejecutando `python mysite/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 mysite/manage.py inspectdb`. Esto examinará las tablas en la base de datos `DATABASE_NAME` e imprimirá para cada tabla el modelo de clase generado. Hecha 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 mysite/manage.py inspectdb > mysite/myapp/models.py
   ```

6. Edita el archivo `mysite/myapp/models.py` para limpiar los modelos generados y realiza cualquier personalización necesaria. Te daremos algunas sugerencias para esto en la siguiente sección.

16.1.2. 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:

1. 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`.
2. 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, querrás 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 agregará *nuevos* registros a estas tablas. El comando `inspectdb` no puede detectar si un campo es autoincrementado, así que esta en ti cambiar esto a `AutoField`, si es necesario.
3. 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).
4. 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.
5. 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 `Book` tiene una `ForeignKey` al modelo `Author`, el modelo `Author` debe ser definido antes del modelo `Book`. Si necesitas crear una relación en un modelo que todavía no esta definido, puedes usar el nombre del modelo, en vez del objeto modelo en sí.
6. `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`.
7. 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`.

16.2. 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.

16.2.1. 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 [20].

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.

16.2.2. 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):
        # Check the username/password and return a User.
```

Pero podría tambien autentificar un token, como se muestra a continuación:

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a User.
```

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 esta 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):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name, and a hash of the password. For example:

    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:
                # Create a new user. Note that we can set password
                # to anything, because it won't be checked; the password
                # from settings.py will.
                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
```

16.3. 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 el Capítulo 20 cubre el despliegue con Django en Apache/mod_python, por lo tanto tal vez valga la pena leer ese capítulo primero 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 20 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 mysite.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 mysite.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/`.

16.4. ¿Qué sigue?

Hablando del sitio de administración de Django y sobre cómo acomodar el framework para encajar con necesidades existentes, otra tarea común es personalizar el sitio de administración de Django. El **'próximo capítulo'_** se enfoca en dicha personalización.

Duplicate explicit target name: “próximo capítulo”.

[19]N. del T.: del inglés “legacy databases and applications”, aplicaciones y base de datos que ya están en uso en entornos de producción.
[20]N. del T.: del inglés “Python path”.

Capítulo 17

Extendiendo la Interfaz de Administración de Django

El Capítulo 6 introdujo la interfaz de administración de Django, y ya es tiempo de volver atrás y dar una mirada más minuciosa al asunto.

Como dijimos varias veces antes, la interfaz de administración es una de las características más sobresalientes de este framework, y la mayoría de los desarrolladores que usan Django lo encuentran útil y eficiente. Debido a que esta interfaz es tan popular, es comun que los desarrolladores quieran personalizarlo o extenderlo.

Las ultimas secciones del Capítulo 6 ofrecieron algunas maneras simples de personalizar ciertos aspectos de la interfaz. Antes de continuar con este capitulo, considera revisar ese material; cubre como personalizar las listas de cambio y los formularios de edición de, asi como una forma fácil de "remarcar" la interfaz para que se indentifique con tu sitio.

El capitulo 6 discute también cuando y porqué querrías usar la interfaz de administración y desde hicimos un gran salto desde esos párrafos hasta este punto, lo reproduciremos nuevamente aquí:

> Obviamente, es muy útil para modificar datos (se veía venir). Si tenemos cualquier tipo de tarea de introducción de datos, el administrador es lo mejor que hay. 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 caracteristica, después de todo. En el periódico donde Django fue creado orginalmente, el desarrollo una característica tipica online --un reporte especial sobre la calidad del agua del acueducto municipal, pongamos-- implicaba algo así:
>
> - El periodista responsable del artículo se reunecon 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 enforcarse en desarrollar la interfaz accesible publicamente (¡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 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.

La interfaz de administración de Django maneja estos casos comunes con algunas o ninguna personalización. Aunque, como sucede con la mayoría de las generalizaciones en el diseño, la gestión unificada de todos estos casos significa que la interfaz no maneja igual de bien otros modos de edición.

Hablaremos de estos casos para los que la interfaz de administración de Django *no está* diseñada un poquito más adelante, pero primero, vayamos a una breve disgreción para una discusión filosófica.

17.1. El Zen de la aplicación Admin

En su núcleo, la interfaz de administración de Django está diseñada para una sola actividad:

> Usuarios confiables editando contenido estructurado.

Sí, es extremadamente simple -- pero esa simplicidad se basa en un montón de asunciones importantes. La entera filosofía de la interfaz de administración de Django sigue directamente estas a asunciones , así que vamos a cavar sobre el subtexto de esta frase en la secciones que siguen.

17.1.1. "Usuarios confiables ..."

La intefaz de administración está diseñada para ser usada por usuarios en lo que tú, el desarrollador, *confías*. Esto no significa sólo "gente que ha sido autenticada"; significa que Django asume que se puede confiar que tus editores de contenido harán las cosas correctas.

Significa además que no hay procesos de aprobación para la edición de contenido -- si confias en tus usuarios, nadie necesita aprobar sus ediciones. Otra implicancia es que el sistema de permisos, aunque poderoso, no permite (al momento en el que se escribe esto) limitar accesos basados en objetos específicos. Si confias en que alguien edite sus propias historias, confias en que ese usuario no edite las historias de cualquier otro sin permiso.

17.1.2. "... editando ..."

El propósito primario de la interfaz de administración de Django es dejar que la gente edite información. Esto parece obvio al principio, pero de nuevo tiene poderosas y profundas repercusiones.

Por ejemplo, aunque la interfaz es bastante util para revisar datos (según se ha descripto=, no está diseñada con este propósito en mente. Por caso, nota la ausencia de un permiso "puede ver" (ve el Capítulo 12). Django asume que si la gente puede ver el contenido en la interfaz de administraión, entonces tambien tienen permito editarlo.

Otra cosa más importante es la ausencia de cualquier cosa que se aproxime remotamente a un "flujo de trabajo". Si una tarea dada requiere una serie de pasos, no existe algo que permita forzar a que estos pasos se realicen en un determinado orden. La interfaz se concentra en *editar*, no en actividades alrededor de la edición. Esta supresión de un flujo de trabajo tambien proviene del principio de confianza: la filosofía de la interfaz es que este flujo es una decisión personal, no algo que se pueda implementar en código.

Finalmente, nota la ausencia de agregaciones en la interfaz. Esto es, no existe la infraestructura necesaria para mostrar totales, promedios y esas cosas. De nuevo, la interfaz es para editar -- y se espera que escribas tus vistas personalizadas para todo el resto.

17.1.3. "... contenido estructurado"

Como el resto de Django, la interfaz prefiere que trabajes con datos estructurados. Esto es porque sólo sirve para editar información almacenada en modelos de Django; para cualquier otra cosa, como datos almacenados en archivos, necesitarás vistas propias.

17.1.4. Parada Completa

A esta altura debería estar claro que la interfaz de administración de Django *no* intenta ser todas las cosas para toda la gente; y en cambio, elegimos enfocarnos en una cosa y hacerla extremadamente bien.

Cuando se va a extender la interfaz de administración, mucha de esa misma filosofía se sostiene (nota que "extensibilidad" no figura de nuestros objetivos). Debido a que vistas personalizadas pueden hacer *cualquier cosa*, y debido a que estas puede ser visualmente integradas a la interfaz de administración muy facilmente (como se describe en la siguiente sección), las posibilidades de personalización incorporadas están un poco limitadas por diseño.

Deberías tener en mente que la interfaz de administración es "sólo una aplicación"; y aunque sea una muy compleja, no hace nada que cualquier desarrollador Django con suficiente tiempo no podría reproducir. Es enteramente posible

que en el futuro alguien desarrolle una interfaz de adminitración diferente que esté basada en un conjunto de asunciones distintas y que por lo tanto se comportará de otra manera.

Finalmente, debemos destacar que, a la fecha que escribimos esto, los desarrolladores de Django trabajaban en una nueva versión de la interfaz de administración que permite mucha más flexibilidad y personalización. Para el momento en que leas esto, esas nuevas caracteristicas pudieron haberse incorporado a la distribución de Django oficial. Para averiguar al respecto, preguntale a alguien de la comunidad Django si la rama "newforms-admin" ha sido integrada.

17.2. Pesonalizando las plantillas de la interfaz

Como sale de fábrica, Django provee un número de herramientas para personalizar las plantillas de la interfaz que vienen integradas, las cuales veremos pronto, pero para las tareas detrás de ellas (por ejemplo, cualquier cosa que requiera un flujo de trabajo específico o permisos granulares), necesitarás leer la sección titulada "Creando Vistas de administración personalizadas", más adelante en este capítulo.

Para ahora, miremos algunas maneras rápidas de modificar el aspecto (y, en cierto grado, el comportamiento) de la interfaz de administración. El Capítulo 6 cubre algunas de las tareas más comunes: "cambiar la marca" de la interfaz de adminitración (para todos esos "Jefes Pelopunta" que odian el azul) y proveer un formulario de administración personalizado.

Pasado ese punto, el objetivo usualmente implica cambiar alguna de las plantillas para un item en particular. Cada vista de administración -- las listas de cambio, los formularios de edición, las páginas de confirmación de eliminación y vistas de historial -- tienen una plantilla asociada que puede ser reescrita de diferentes maneras.

Primero, puedes reescribir la plantilla globalmente. La vista de administración busca plantillas utilizando el mecanismo de carga de plantillas estándar, por lo que si creas tus plantillas en alguno de los directorios declarados para tal fin, Django cargará esas en vez de las vienen por defecto. Estas plantillas globales se describen el la Tabla 17-1.

Cuadro 17.1: Plantillas globales de la interfaz de administración

Vista	Nombre de la plantilla base
Lista de cambios	`admin/change_list.html`
Formulario para agregar/editar	`admin/change_form.html`
Confirmación de eliminación	`admin/delete_confirmation.html`
Historial de un objeto	`admin/object_history.html`

La mayoría de las veces, sin embargo, querrás cambiar la plantilla sólo para un único objeto o aplicación (no globalmente). Así, cada vista busca primero plantillas para modelos y aplicaciones específicas, en el siguiente orden:

- `admin/<aplicación>/<nombre_objeto>/<plantilla>.html`
- `admin/<aplicación>/<plantilla>.html`
- `admin/<plantilla>.html`

Por ejemplo, la vista del formulario de agregar/editar para un modelo `Libro` in la aplicación `libros` busca plantillas en este orden:

- `admin/libros/libro/change_form.html`
- `admin/libros/change_form.html`
- `admin/change_form.html`

17.2.1. Plantillas de modelos propios

La mayoría de las veces, y querrás usar la primer plantilla para crear una basada destinada a un modelo específico. Usualmente la mejor forma de realizar esto es extendiendo y agregando información a uno de los bloques definidos en la plantilla que se está modificando.

Por ejemplo, supongamos que queremos agregar un pequeño texto de ayuda en la cabecera de nuestra página de libros. Quizas algo parecido a lo que muestra la Figura 17-1.

Esta es una manera muy fácil de hacerlo: simplemente crea una plantilla llamada `admin/libreria/libro/change_form.ht` e inserta este código:

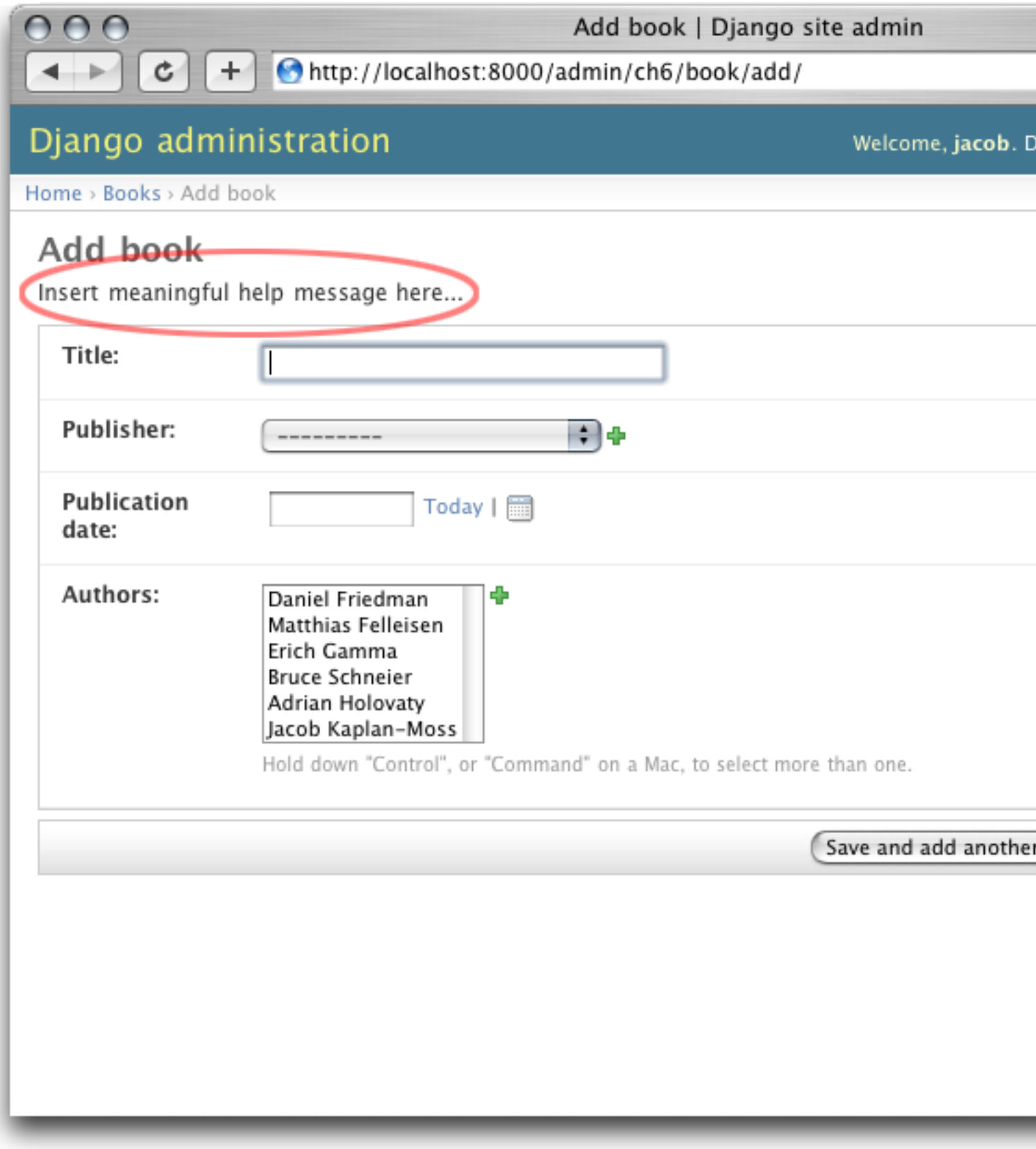

Figura 17.1: Un formulario de edición de libros personalizado

```
{% extends "admin/change_form.html"%}

{% block form_top%}
  <p>Inserta un mensaje de ayuda significativo aquí...</p>
{% endblock%}
```

Todas esta plantillas definen un número de bloques que puedes sobreescribir. Como con la mayoría de los programas, la mejor documentación es el propio código, por lo que te animamos a mirar las plantillas originales (que se encuentran en `django/contrib/admin/templates/`) para trabajar con la información más actualizada.

17.2.2. JavaScript Personalizado

Un uso común para estas plantillas propias para modelos

A common use for these custom model templates implica agregar código JavaScript extra a las páginas de la interfáz -- posiblemennte para implementar algún widget especial o un comportamiento del lado del cliente.

Por suerte, esto no podría ser más fácil. Cada plantilla del administrador define un `{% block extrahead%}`, el cual puedes usar para incluir contenido extra dentro del elemento `<head>`. Por ejemplo, incluir la biblioteca jQuery (http://jquery.com/) en tu página de historia de objetos, es tan simple como esto:

```
{% extends "admin/object_history.html"%}

{% block extrahead%}
    <script src="http://media.ejemplo.com/javascript/jquery.js" type="text/javascript"></script>
    <script type="text/javascript">

        // el código que utiliza jQuery iría aquí...

    </script>
{% endblock%}
```

> **Nota**
>
> No estamos seguros porqué necesitarías jQuery en la página de historia de objetos, pero, por supuesto, este ejemplo es válido para cualquier plantilla de la interfaz de administración.

Puedes usar esta técnica para incluir cualquier tipo de controladores JavaScript que puedas necesitar en tus formularios.

17.3. Creando vistas de administración personalizadas

Hasta ahora, cualquiera que haya buscando agregar *comportamientos* personalizados a la interfaz de administración probablemente esté un poco frustrado. "Todo lo que han dicho es cómo cambiar la interfaz *visualmente*", los escuchamos llorar. "¿Pero como puedo cambiar la forma en que la interfaz de administración *funciona*?"

La primer cosa para entender es que *esto no es mágico*. Esto es, nada de lo que la interfaz hace es *especial* de manera alguna -- ya que se trata simplemente de un conjunto de vistas (que se encuentran en `django.contrib.admin.views`) que manipulan datos como cualquier otra vista.

Seguro, hay bastante código allí, y se debe a que tienen que lidear con todas las opciones, diferentes tipos de campos, y configuraciones que inflyen en el comportamiento. No obstante, cuando te das cuenta que la intefaz de administración es sólo un juego de vistas, agregar las tuyas propias es más fácil de entender.

A modo de ejemplo, agreguemos una vista "reporte de editores" a nuestra aplicación de libros del Capítulo 6. Construiremos una vista de administración que muestre la lista de libros en función de los editores -- un ejemplo bastante típico de vista de "reporte" que puedes necesitar construir.

Primero, actualicemos nuestro archivo URLconf. Necesitamos insertar esta línea:

```
(r'^admin/libros/reporte/$', 'misitio.libros.admin_views.reporte'),
```

antes de la línea que incluye las vistas del administrador. Un esqueletos del URLconf puede parecerse a algo así:

```
from django.conf.urls.defaults import *
```

```
urlpatterns = patterns('',
    (r'^admin/libreria/reporte/$', 'libreria.admin_views.reporte'),
    (r'^admin/', include('django.contrib.admin.urls')),
)
```

¿Por qué ponemos la vista personalizada *antes* de incluir las del administrador? Recuerda que Django procesa los patrones de URL en orden. La inclusión de los patrones de urls del administrador coincide con casi cualquier cosa que llega a su punto de inclusión, por lo que si invertimos el orden de esas lineas, Django encontrará una vista por omisión para ese patrón y no funcionará como queremos. En este caso particular, intentará cargar un un lista de cambios para un modelo "Reporte" en la aplicación "libros", que no existe.

Ahora escribamos nuestra vista. Para hacer honor a la simplicidad, sólo cargaremos todos los libros dentro del contexto, y dejaremos que la plantilla maneje el agrupamiento con la etiqueta `{% regroup %}`. Crea un archivo `books/admin_views.py`, con este código:

```
from misitio.libros.models import Libro
from django.template import RequestContext
from django.shortcuts import render_to_response
from django.contrib.admin.views.decorators import staff_member_required


def reporte(request):
    return render_to_response(
        "admin/libros/reporte.html",
        {'lista_libros' : Book.objects.all()},
        RequestContext(request, {}),
    )
reporte = staff_member_required(reporte)
```

Debido a que dejamos el agrupamiento a la plantilla, esta vista es bastante simple. Sin embargo, hay algunos fragmentos sutiles dignos de explicitar:

- Usamos el decorador `staff_member_required` de `django.contrib.admin.views.decorators`. Esto es similar a `login_required` discutido en el Capítulo 12, pero este decorador también verifica que el usuario esté marcado como un mientro del "staff", y tenga en consecuencia acceso a la interfaz de administración

  Este decorador protege todos las vistas predefinidas del administrador, y hace que la lógica de autenticación para tus vistas coincida con la del resto de la interfaz.
- Renderizamos una plantilla que se encuntra bajo `admin/`. Aunque esto no estrictamente requerido, se considera una buena practica para mantener todas tus plantillas de administración agrupadas en un directorio `admin/`. También pusimos la plantilla en un directorio llamado `libros` luego de nuestra aplicación -- lo que es también una buena práctica.
- Usamos `RequestContext``como el tercer parámetro (``context_instance`) para `render_to_response`. Esto asegura que la información sobre el usuario en curso está disponible para la plantilla.

  Mira el **`Capítulo 10`_** para saber más sobre `RequestContext`.

Finalmente, haremos una plantilla para esta vista. Extenderemos una plantilla de la administración para que lograr que nuestra vista coincida visualmente con el resto de la interfaz:

```
{% extends "admin/base_site.html"%}

{% block title%}Lista de libros por editor{% endblock%}

{% block content%}
<div id="content-main">
  <h1>Lista de libros por editor:</h1>
  {% regroup lista_libros|dictsort:"editor.nombre" by editor as libros_por_editor%}
  {% for editor in libros_por_editor%}
    <h3>{{ editor.grouper }}</h3>
    <ul>
      {% for libro in editor.list|dictsort:"titulo"%}
```

```
            <li>{{ libro }}</li>
          {% endfor%}
        </ul>
      {% endfor%}
    </div>
    {% endblock%}
```

Al extender `admin/base_site.html`, conseguimos el *look and feel* de la interfaz de administración de Django "gratis". La Figura 17-2 muestra como luce el resultado.

Puedes usar esta técnica para agregar cualquier cosa que sueñes para la interfaz de administración. Recuerda que las llamadas vistas de administración personalizadas en realidad son sólo vistas comunes de Django; por lo que puedes usar todas las técnicas aprendidas en el resto de este libro para proveer una interfaz con tanta complejidad como necesites.

Cerraremos este capítulo con algunas ideas para vistas de administración personalizadas.

17.4. Sobreescribiendo vistas incorporadas

Algunas veces las vistas de administración por omisión simplemente no te sirven. Fácilmente puedes reemplazarlas por las tuyas propias en cualquier etapa de la interfaz de administración; simplemente haz que tu URL "haga sombra* sobre la que incorporada. Es decir, si tu vista viene antes que la vista incorporada de la aplicación en URLconf, tu vista será invocada por sobre la de omisión.

Por ejemplo, podriamos reemplazar la vista incorporada "para crear" libros con un formulario que permita a los usuarios ingresar simplemente un código ISBN. Luego podriamos buscar la información del libro desde `http://isbn.nu` y crear el objeto automáticamente.

El código para esa vista te lo dejamos como ejercicio, pero la parte importante esta partecita del URLconf:

```
(r'^admin/libreria/libro/add/$', 'misitio.libros.admin_views.agregar_por_isbn'),
```

Si esta linea aparece antes que las URLs de administracion en tu URLconf, la vista `agregar_por_isbn` reemplazará completamente a la vista estándar para ese modelo.

Podriamos seguir un truco similar para reemplazar la página de confirmación de eliminación, la de edición o cualquier otra parte de la interfaz.

17.5. ¿Qué sigue?

Si tu idioma nativo es el inglés --cosa que gracias a los traductores ya no es necesaria para leer este libro-- quizas no te hayas enterado de una las más fantásticas características de la interfaz de administracion: ¡está disponible en casi 40 idiomas distintos! Esto es posible gracias al framework de internacionalización de Django (y el duro trabajo de los traductores voluntarios de Django). El **'próximo capítulo'_** explaya como usar este framework para crear sitios Django localizados.

¡Avanti!

Duplicate explicit target name: "próximo capítulo".

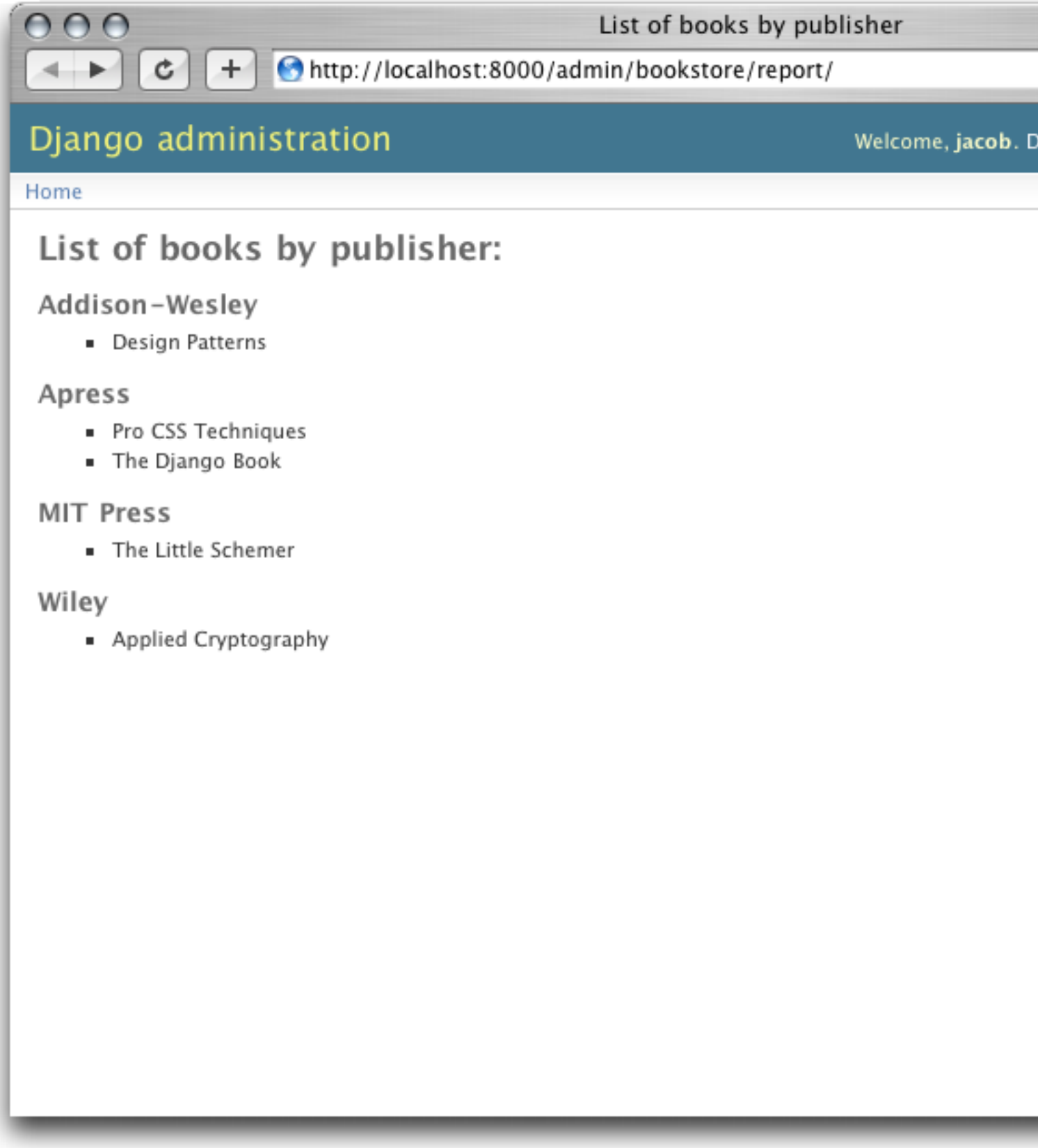

Figura 17.2: Una vista personalizada "libros por editor".

Capítulo 18

Internacionalización

Django fue originalmente desarrollado exactamente en el 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). Como la mayoría de los proyectos open source, sin embargo, la comunidad de Django creció hasta incluir gente de todo el globo. 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.

Internacionalización se refiere al proceso de diseño de programas para el uso potencial de cualquier ***locale*** Esto incluye el marcado del texto (tales como elementos de la interfaz con el usuario o mensajes de error) para su futura traducción, la abstracción de la visualización de fechas y horarios de manera que sea posible respetar diferentes estándares locales, la provisión de lo necesario para admitir 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 omitidos entre la "I" inicial y la "N" final).

Localización se refiere al proceso específico de traducir un programa internacionalizado para su uso en un ***locale*** 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 dependientes del ***locale*** 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 ***hooks*** a tu código Python y a tus plantillas. Esos ***hooks*** 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 esos ***hooks*** 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/) via el módulo estándar `gettext` incluido en Python.

Si no necesitas usar internacionalización:

Los ***hooks*** de internacionalización de Django se encuentran activos por omisión, lo cual incurre en un pequeño ***overhead***. Si no utilizas 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`.

18.1. Especificando 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.

18.1.1. Funciones estándar de traducción

Las cadenas de traducción se especifican usando la función `_()`. (Si, el nombre de la función es el carácter guión bajo). Esta función está disponible globalmente (o sea como un componente incluido); no es necesario que lo importes.

En este ejemplo, el texto `"Welcome to my site."` está marcado como una cadena de traducción:

```
def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

La función `django.utils.translation.gettext()` es idéntica a `_()`. Este ejemplo es idéntico al anterior:

```
from django.utils.translation import gettext
def my_view(request):
    output = gettext("Welcome to my site.")
    return HttpResponse(output)
```

La mayoría de los desarrolladores prefiere usar `_()`, debido a que es más corta.

La traducción funciona también sobre valores computados. Este ejemplo es idéntico a los dos anteriores:

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

La traducción funciona también sobre variables. De nuevo, este es otro ejemplo idéntico:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(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, `make-messages.py`, no será capaz de encontrar esas cadenas. Trataremos `make-messages` más adelante).

Las cadenas que le pasas a `_()` o `gettext()` pueden contener marcadores de posición (por *placeholders*), especificados con la sintaxis estándar de interpolación de cadenas con nombres, por ejemplo:

```
def my_view(request, n):
    output = _('%(name)s is my name.') % {'name': n}
    return HttpResponse(output)
```

Esta técnica permite que las traducciones específicas de cada idioma reordenen el texto de los marcadores de posición. Por ejemplo, una traducción al inglés podría ser `Adrian is my name`, mientras que una traducción al español podría ser `Me llamo Adrian`, con el marcador de posición (el nombre) ubicado a continuación del texto traducido y no antes del mismo.

Por esta razón, deberías usar interpolación de cadenas con nombres (por ejemplo `%(name)s`) en lugar de interpolación posicional (por ejemplo `%s` o `%d`). Si usas interpolación posicional las traducciones no serán capaces de reordenar el texto de los marcadores de posición.

18.1.2. Marcando cadenas como no-op

Usa la función `django.utils.translation.gettext_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.

18.1.3. Traducción perezosa

Usa la función `django.utils.translation.gettext_lazy()` para traducir cadenas en forma perezosa -- cuando el valor es accedido en lugar de cuando se llama a la función `gettext_lazy()`.

Por ejemplo, para marcar el atributo `help_text` de un campo como traducible, haz lo siguiente:

```
from django.utils.translation import gettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=gettext_lazy('This is the help text'))
```

En este ejemplo, `gettext_lazy()` almacena una referencia perezosa a la cadena -- no el 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.

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.utils.translation import gettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

Usa siempre traducciones perezosas en modelos Django (de lo contrario no serán traducidos correctamente para cada usuario). Y es una buena idea agregar también traducciones de los nombres de campos y nombres de tablas. Esto significa escribir las opciones `verbose_name` y `verbose_name_plural` en forma explícita en la clase `Meta`:

```
from django.utils.translation import gettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))
    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('mythings')
```

18.1.4. Pluralización

Usa la función `django.utils.translation.ngettext()` para especificar mensajes que tienen formas singular y plural distintas, por ejemplo:

```
from django.utils.translation import ngettext
def hello_world(request, count):
    page = ngettext(
        'there is%(count)d object',
        'there are%(count)d objects', count
    )% {'count': count}
    return HttpResponse(page)
```

`ngettext` tiene tres argumentos: la cadena de traducción singular, la cadena de traducción plural y el número de objetos (el cual es pasado a los idiomas de traducción como la variable `count`).

18.2. Especificando cadenas de traducción en código de plantillas

Las traducciones en las plantillas 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%}` marca una cadena para su traducción:

```
<title>{% trans "This is the title."%}</title>
```

Si solo deseas marcar un valor para traducción pero para traducción posterior, usa la opción `noop`:

```
<title>{% trans "value" noop%}</title>
```

No es posible usar variables de plantilla en `{% trans%}` -- solo están permitidas cadenas constantes, rodeadas de comillas simples o dobles. Si tu traducción requiere variables (marcadores de posición) puedes usar por ejemplo `{% blocktrans%}`:

```
{% blocktrans%}This will have {{ value }} inside.{% endblocktrans%}
```

Para traducir una expresión de plantilla -- por ejemplo, cuando usas filtros de plantillas -- necesitas asociar la expresión a una variable local que será la que se usará dentro del bloque de traducción:

```
{% blocktrans with value|filter as myvar%}
  This will have {{ myvar }} inside.
{% endblocktrans%}
```

Si necesitas asociar más de una expresión dentro de una etiqueta `blocktrans`, separa las partes con `and`:

```
{% blocktrans with book|title as book_t and author|title as author_t%}
  This is {{ book_t }} by {{ author_t }}
{% 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%}
  There is only one {{ name }} object.
{% plural%}
  There are {{ counter }} {{ name }} objects.
{% endblocktrans%}
```

Internamente, todas las traducciones en bloque y en línea usan las llamadas apropiadas a `gettext`/`ngettext`.

Cuando usas `RequestContext` (ver **'Capítulo 10'_**), 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 "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).

Puedes también cargar los siguientes valores usando etiquetas de plantilla:

```
{% load i18n%}
{% get_current_language as LANGUAGE_CODE%}
{% get_available_languages as LANGUAGES%}
{% get_current_language_bidi as LANGUAGE_BIDI%}
```

También existen ***hooks*** de traducción que 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 una cadena de traducción, por ejemplo:

```
{% some_special_tag _("Page not found") value|yesno:_("yes,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.

18.3. Creando archivos de idioma

Una vez que hayas etiquetado tus cadenas para su posterior traducción, necesitas escribir (u obtener) las traducciones propiamente dichas. En esta sección explicaremos como es que eso funciona.

18.3.1. 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 tiene 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 cuales son los idiomas actualmente incluidos.

El script debe ser ejecutado desde una de tres ubicaciones:

- El directorio raíz `django` (no una copia de trabajo de Subversion, sino el que se halla referenciado por `$PYTHONPATH` o que se encuentra en algún punto debajo de esa ruta.
- El directorio raíz de tu proyecto Django
- El directorio raíz de tu aplicación Django

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 `conf/locale`. En el ejemplo `de`, el archivo será `conf/locale/de/LC_MESSAGES/django.po`.

Si es ejecutado sobre el árbol de tu proyecto o tu aplicación, hará lo mismo pero la ubicación del directorio locale es `locale/LANG/LC_MESSAGES` (nota que no tiene un prefijo `conf`). La primera vez que lo ejecutes en tu árbol necesitarás crear el directorio `locale`.

> **¿Sin gettext?**
>
> Si no tienes instaladas las utilidades `gettext`, `make-messages.py` 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.

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 traducciones al idioma en cuestión propiamente dichas.

Por ejemplo, si tu aplicación Django contiene una cadena de traducción para el texto `Welcome to my site`:

```
_("Welcome to my site.")
```

entonces `make-messages.py` habrá creado un archivo `.po` que contendrá el siguiente fragmento -- un mensaje:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
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 que mantienes 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 via 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."
```

Notar 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:

```
make-messages.py -a
```

18.3.2. 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 `bin/compile-messages.py`.

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 `make-messages.py`, ejecuta `compile-messages.py` de la siguiente manera:

```
bin/compile-messages.py
```

Y eso es todo. Tus traducciones están listas para ser usadas.

18.4. 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.middleware.common.CommonMiddleware',
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware'
)
```

`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.
- Se 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.transla` 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:

```
_ = lambda s: s

LANGUAGES = (
      ('de', _('German')),
      ('en', _('English')),
)
```

Con este esquema, `make-messages.py` 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 traducciones para tu aplicación que no se encuentran en el conjunto de traducciones incluidas en el código fuente de Django, querrás proveer al menos traducciones 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 traducciones 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 hello_world(request, count):
    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`.

18.5. 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:

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

(Nota que este ejemplo publica la vista en `/i18n/setlang/`).

La vista espera ser llamada vía el método `GET`, con un parámetro `language` incluido en la cadena de consulta. Si el soporte para sesiones está activo, la vista guarda la opción de idioma en la sesión del usuario. Caso contrario, guarda el idioma en una cookie `django_language`.

Después de haber fijado la opción de idioma Django redirecciona al usuario, para eso sigue el siguiente algoritmo:

- Django busca un parámetro `next` en la cadena de consulta.
- 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 un último recurso.

Este es un fragmento de código de plantilla HTML de ejemplo:

```
<form action="/i18n/setlang/" method="get">
<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>
```

18.6. Usando traducciones en tus propios proyectos

Django busca traducciones 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 su propias traducciones, y puedes reemplazar traducciones 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 traducciones en un gran archivo de mensajes. Es tu elección.

> **Nota**
>
> Si estás fijando 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 fijando manualmente tus variables de configuración dicho archivo no existe).

Todos los repositorios de archivos de mensajes están estructurados de ka misma manera:

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- Todas las rutas listandas en `LOCALE_PATHS` en tu archivo de configuración son examinadas en ese orden en búsquda 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 `make-messages.py` 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 `compile-messages.py` para producir los archivos binarios `django.mo` usados por `gettext`.

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 proeyctos, posiblemente quieras usar traducciones específicas a dichas aplicaciones. Pero el usar traducciones específicas a aplicaciones y aplicaciones en proyectos podrían producir problemas extraños con `make-messages.py`. `make-messages` 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 traducciones) fuera del árbol del proyecto. De esa forma *make-messages.py*“ ejecutado a nivel proyecto sólo traducirá cadenas que están conectadas a tu proyecto y no cadenas que son distribuidas en forma independiente.

18.7. Traducciones y JavaScript

El agregar traducciones 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 esos problemas: convierte las traducciones a JavaScript, de manera que puedas llamar a `gettext` y demás desde JavaScript.

18.7.1. 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 `info_dict` o en la URL.

La forma de usar esto es asi:

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = patterns('',
    (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
)
```

Cada cadena en `package` debe seguir la sintaxis paquete 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 especificas múltiples paquetes, todos esos catálogos son fusionados en un catálogo único. 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 = patterns('',
    (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`.

18.7.2. Usando el catálogo de traducciones 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 traducciones 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('this is to be translated'));
```

Existe incluso una interfaz `ngettext` y una función de interpolación de cadenas:

```
d = {
    count: 10
};
s = interpolate(ngettext('this is%(count)s object', 'this are%(count)s objects', d.count), d);
```

La función `interpolate` admite tanto interpolación posicional como interpolación con nombres. De manera que el código de arriba podría haber sido escrito de la siguiente manera:

```
s = interpolate(ngettext('this is%s object', 'this are%s objects', 11), [11]);
```

La sintaxis de interpolación se tomó prestada de Python. 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).

18.7.3. Creando catálogos de traducciones JavaScript

Los catálogos de traducciones se crean y actualizan de la misma manera que el resto de los catálogos de traducciones de Django: con la herramienta `make-messages.py`. La única diferencia es que es necesario que proveas un parámetro `-d djangojs`, de la siguiente forma:

```
make-messages.py -d djangojs -l de
```

Esto crea o actualiza el catálogo de traducción para JavaScript para alemán. Luego de haber actualizado catálogos, sólo ejecuta `compile-messages.py` de la misma manera que lo haces con los catálogos de traducción normales de Django.

18.8. Notas para usuarios familiarizados con gettext

Si conoces `gettext` podrías notar las siguientes particularidades en la forma en que Django maneja las traducciones:

- 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 traducciones globales. El dominio `djangojs` se usa sólo para catálogos de traducciones 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 ***wrappers*** Python alrededor de `xgettext` y `msgfmt`. Esto es más que nada por conveniencia.

18.9. ¿Qué sigue?

Este capítulo esencialmente concluye nuestra cobertura de las características de Django. Deberías conocer lo suficiente para comenzar a producir tus propios sitios usando Django.

Sin embargo, escribir el código es solo el primer paso de la instalación de un sitio Web exitoso. Los siguientes dos capítulos cubren las cosas que necesitarás conocer si deseas que tu sitio sobreviva en el mundo real. El **'Capítulo 19'_** trata cómo puedes hacer para hace que tus sitios y tus usuarios estén seguros ante atacantes maliciosos y el Capítulo 20 detalla cómo instalar una aplicación Django en uno o varios servidores.

Capítulo 19

Seguridad

Internet puede ser un lugar aterrador.

En estos tiempos, los papelones de seguridad con alta exposición pública parecen ser cosa de todos los días. Hemos visto virus propagarse con una velocidad asombrosa, ejércitos de computadoras comprometidas ser empuñados 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.

19.1. El tema de la seguridad en la Web

Si aprendes sólo una cosa de este capítulo, que sea esto:

```
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 con igual facilidad podría tratarse de un vil cracker buscando un 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 donde vienen estos datos¿'

19.2. 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
    # execute the SQL here...
```

> **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 todas 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?

19.2.1. 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 foos 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])
    # ... do something with the results
```

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.

19.3. Cross-Site Scripting (XSS)

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":

```
def say_hello(request):
    name = request.GET.get('name', 'world')
    return render_to_response("hello.html", {"name" : name})
```

Esta vista simplemente lee un nombre desde un parámetro `GET` y pasa dicho nombre a la plantilla `hello.html`. Podríamos escribir una plantilla para esta vista de la siguiente manera:

```
<h1>Hello, {{ name }}!</h1>
```

De manera que si accediéramos a `http://example.com/hello/?name=Jacob`, la página renderizada contendría lo siguiente:

```
<h1>Hello, Jacob!</h1>
```

Pero atención -- ¿qué sucede si accedemos a `http://example.com/hello/?name=<i>Jacob</i>`? En ese caso obtenemos esto:

```
<h1>Hello, <i>Jacob</i>!</h1>
```

Obviamente, un atacante no usará algo tan inofensivo como etiquetas `<i>`; 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 aun 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 agregana 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.

19.3.1. La solución

La solución es simple: *siempre escapa todo* el contenido que pudiera haber sido enviado por un usuario. Si simplemente reescribiéramos nuestra plantilla de la siguiente manera:

```
<h1>Hello, {{ name|escape }}!</h1>
```

ya no seríamos vulnerables. Debes usar *siempre* la etiqueta `escape` (o algo equivalente) cuando visualizas en tu sitio contenido enviado por el usuario.

¿Porqué simplemente Django no hace esto por mí?

Modificar Django para que *escape* automáticamente todas las variables visualizadas en plantillas es un tópico de frecuente tratamiento en la lista de correo de desarrollo de Django.

Hasta ahora, las plantillas Django han evitado este comportamiento debido a que esto cambia sutilmente algo que debería ser un comportamiento no complejo (la visualización de variables). Es un asunto no trivial y una decisión de compromiso difícil de evaluar. Agregando comportamiento implícito y oculto va contra los ideales de base de Django (y los de Python), pero la seguridad es igual de importante.

Todo esto nos lleva, entonces, a afirmar que es muy probable que Django incorpore alguna forma de comportamiento de *auto-escaping* (o algo cercano a *auto-escaping*) en el futuro. Es una buena idea revisar la documentación oficial de Django para conocer las novedades respecto a las características de Django; esta será siempre más actual que este libro, especialmente que la versión impresa.

Aun si Django incorpora esta característica *debes* formar el hábito de preguntarte, en todo momento, "¿De donde provienen estos datos¿'. Ninguna solución automática protegerá tu sitio de ataques XSS el 100 % del tiempo.

19.4. Cross-Site Request Forgery

La Cross-site request forgery (CSRF) (Falsificación de peticiones inter-sitio) sucede 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 14.

19.5. 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. El Capítulo 12 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 simple 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/?PHPSE`
  Un atacante que logre engañar a un usuario para que haga click 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 click 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.

19.5.1. 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 12) 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.

19.6. 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 caracter 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.

19.6.1. 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.

19.7. Directory Traversal

Directory traversal 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 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 la 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 *escaping* 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!

19.7.1. 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.

> **Nota**
>
> No es necesario decirlo, *¡nunca* debes escribir código que pueda leer cualquier área del disco!

Un buen ejemplo de cómo hacer este *escaping* yace en la vista de publicación de contenido 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.

19.8. 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.

19.8.1. La solución

Django tiene 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 mod_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.

19.9. Palabras finales sobre la seguridad

Esperamos que toda esta 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. En efecto, siempre es una buena idea dedicar algún tiempo semanalmente o mensualmente a investigar y mantenerse 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.

19.10. ¿Qué sigue?

En el **‘próximo capítulo‘**_, finalmente trataremos las sutilezas de la implementación de Django: como lanzar un sitio de producción y como dotarlo de escalabilidad.

Duplicate explicit target name: ”próximo capítulo“.

Capítulo 20

Implementando Django

A lo largo de este libro, hemos mencionado algunos objetivos que conducen el desarrollo de Django. Facilidad de uso, amigabilidad para nuevos programadores, abstracción de tareas repetitivas -- todos esas metas marcaron el camino de los desarrolladores.

Sin embargo, desde la concepción de Django, ha existido siempre otro objetivo importante: Django debería ser fácil de implementar, y debería poder servir una gran cantidad de tráfico con recursos limitados.

Las motivaciones para este objetivo se vuelven evidentes cuando observas el trasfondo de Django: un pequeño periódico en Kansas difícilmente pueda costear hardware de servidor de última tecnología, por lo que los desarrolladores originales de Django trataron de extraer el máximo desempeño posible de los escasos recursos disponibles. De hecho, por años los desarrolladores de Django actuaron como sus propios administradores de sistema -- ya que simplemente no había suficiente harware como para *necesitar* administradores dedicados a esa tarea -- incluso manejando sitios con decenas de millones de entradas por día.

Como Django se volvió un proyecto open source, este enfoque en el desempeño y la facilidad de implementación se tornó importante por diferentes razones: los desarrolladores aficionados tienen los mismos requerimientos. Los individuos que quieren usar Django están encantados de saber que pueden hospedar un sitio con tráfico entre pequeño y mediano por menos de u$s 10 mensuales.

Pero ser capaz de escalar hacia abajo es solamente la mitad de la batalla. Django también debe ser capaz de escalar *hacia arriba* para conocer las necesidades de grandes empresas y corporaciones. Aquí, Django adopta una filosofía común entre los grupos de software del tipo LAMP que suele llamarse *shared nothing* (nada compartido).

¿Qué es LAMP?

El acrónimo LAMP fue originalmente acuñado para describir un conjunto de software open source utilizado para propulsar muchos sitios web:

- Linux (sistema operativo)
- Apache (servidor web)
- MySQL (base de datos)
- PHP (lenguaje de programación)

A lo largo del tiempo, el acrónimo se ha vuelto más una referencia a la filosofía de este tipo de agrupamiento de software que a cualquiera de estos en particular. Por ello, aunque Django usa Python y es *agnóstico* respecto al motor de base de datos a utilizar, las filosofías probadas por los agrupamientos tipo LAMP permanece en la mentalidad de implementación de Django.
Han habido algunos (más que nada cómicos) intentos de acuñar acrónimos similares para describir los agrupamientos de tecnología que usa Django. Los autores de este libro están encariñados con LAPD (Linux, Apache, PostgreSQL, y Django) o PAID (PostgreSQL, Apache, Internet, y Django). Usa Django y consigue un PAID! (N. de T.: En inglés, PAID significa *pago*).

20.1. Nada Compartido

Esencialmente, la filosofía *shared nothing* se trata de el acoplamiento débil aplicado a todo el conjunto de software utilizado. Esta arquitectura se presentó como respuesta directa a la que en su momento prevalecía: una aplicación de

servidor web monolítica que encapsulaba el lenguaje, la base de datos y el servidor web -- e incluso partes del sistema operativo -- un único proceso (por ejemplo, Java).

Cuando llega el momento de escalar, esto puede ser un problema serio; es casi imposible separar el trabajo de un proceso monolítico entre muchas maquinas físicas diferentes, por lo que las aplicaciones monolíticas requieren servidores enormemente potentes. Estos servidores, por supuesto, cuestan decenas o a veces centenas de miles de dolares, dejando a los sitios web de gran escala lejos del alcance de individuos o pequeñas compañías con buenas ideas pero sin efectivo.

No obstante, lo que la comunidad LAMP descubrió fue que si se separa cada pieza de esa pila de software Web en componentes individuales, se podría fácilmente comenzar con un servidor barato y simplemente ir agregando más servidores baratos a medida que se crece. Si un servidor de base de datos de u$s3000 ya no puede manejar la carga, sencillamente se compraría un segundo (o un tercero, o un cuarto) hasta que pueda. Si se necesita más capacidad de almacenamiento, se agregaría un servidor NFS.

Aunque para que esto funcione, las aplicaciones Web deben dejar de asumir que el mismo servidor es el que maneja cada petición -- o incluso las distintas partes de una petición. En una implementación LAMP (y Django) de gran escala, ¡más de media docena de servidores pueden estar involucrados en servir una sola petición! Las repercusiones a esta situación son numerosas, pero pueden reducirse a estos puntos:

- *El estado no puede ser guardado localmente.* En otras palabras, cualquier dato que deba estar disponible entre múltiples solicitudes, debe almacenarse en algún tipo de almacenamiento permanente como la base de datos o una caché centralizada.
- *El software no puede asumir que los recursos son locales.* Por ejemplo, la plataforma web no puede asumir que la base de datos corre en el mismo servidor; por lo que debe ser capaz de conectarse a servidor de base de datos remoto.
- *Cada pieza del conjunto debe ser fácilmente trasladable o reemplazable.* Si Apache por alguna razón no funciona para la implementación dada, deberías ser posible cambiarlo por otro servidor con mínimas complicaciones. O a nivel hardware, si un servidor web falla, debería ser posible reemplazarlo por otra maquina con ínfimos tiempos de caída. Recuerda, esta filosofía sobre implementación se basa enteramente en hardware barato. Fallas en maquinas individuales deben estar contempladas.

Como probablemente esperabas, Django maneja todo esto más o menos de forma transparente -- ninguna parte de Django viola estos principios -- pero conocer la filosofía ayuda cuando es tiempo de escalar.

¿Pero esto funciona?

Esta filosofía puede sonar bien en papel (o en tu pantalla), pero ¿funciona realmente?
Bueno, en vez de responder directamente, permítanos mostrarle una lista no muy exhaustiva de compañías que han basado sus negocios en esta arquitectura. Probablemente reconozca algunos de estos nombres:

- Amazon
- Blogger
- Craigslist
- Facebook
- Google
- LiveJournal
- Slashdot
- Wikipedia
- Yahoo
- YouTube

Parafraseando la famosa escena de *Cuando Harry conoció a Sally*...: "¡Tendremos lo que ellos tienen¡'

20.2. Un nota sobre preferencias personales

Antes de entrar en detalles, un rápido comentario.

El open source es famoso por sus llamadas guerras religiosas; mucha tinta (digital) ha sido despilfarrada argumentando sobre editores de textos (`emacs` versus `vi`),sistemas operativos (Linux versus Windows versus Mac OS), motores de base de datos (MySQL versus PostgreSQL), y -- por supuesto -- lenguajes de programación.

Nosotros tratamos de permanecer lejos de esas batallas. Simplemente no hay tiempo suficiente.

Sin embargo, hay algunas elecciones que tomar al momento de implementar Django, y constantemente nos preguntar por nuestras preferencias. Conscientes de que explicitar esas preferencias puede encender una de de esas batallas ya mencionadas, la mayoría de las veces hemos tratado de evitarlo. Pero para permitir un debate completo lo explicitaremos aquí. Preferimos lo siguiente:

- Linux (específicamente Ubuntu) como nuestro sistema operativo
- Apache y mod_python para el servidor web
- PostgreSQL como servidor de base de datos

Por supuesto, podemos indicarles muchos usuarios de Django que han hecho otras elecciones con gran éxito.

20.3. Usando Django con Apache y mod_python

Apache con mod_python es actualmente la configuración más robusta para usar Django en un servidor en producción.

mod_python (http://www.djangoproject.com/r/mod_python/) es un plugin de Apache que embebe Python dentro de Apache y carga código Python en memoria cuando el servidor se inicia. El código permanece en memoria a lo largo de la vida del proceso Apache, lo que repercute en aumentos significativos de desempeño comparado con otros arreglos de servidor.

Django requiere Apache 2.x y mod_python 3.x, y nosotros preferimos el módulo de multiprocesamiento (MPM) prefork de Apache, por sobre el MPM worker.

Nota

Configurar Apache está *claramente* más allá del alcance de este libro, por lo que simplemente mencionaremos algunos detalles que necesitamos. Afortunadamente existen grandes recursos disponibles para aprender más sobre Apache. Algunos de los que nos gustan son los siguientes:

- La documentación gratuita de Apache, disponible via `http://www.djangoproject.com/r/apache/docs/`
- *Pro Apache, Third Edition* (Apress, 2004) de Peter Wainwright, disponible via `http://www.djangoproject.com/r/books/pro-apache/`
- *Apache: The Definitive Guide, Third Edition* (O'Reilly, 2002) de Ben Laurie y Peter Laurie, disponible via `http://www.djangoproject.com/r/books/apache-pra/`

20.3.1. Configuración básica

Para configurar Django con mod_python, primero debe asegurarse de que tiene Apache instalado con el módulo mod_python activado. Esto usualmente significa tener una directiva `LoadModule` en tu archivo de configuración de Apache. Puede parecerse a esto:

```
LoadModule python_module /usr/lib/apache2/modules/mod_python.so
```

Luego, edite su archivo de configuración de Apache y agregue lo siguiente:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE misitio.settings
    PythonDebug On
</Location>
```

Asegurese de reemplazar `misitio.settings` por el `DJANGO_SETTINGS_MODULE` apropiado para tu sitio.

Esto le dice a Apache, "Usa mod_python para cualquier URL en '/' o bajo ella, usando el manejado mod_python de Django". Le pasa el valor de `DJANGO_SETTINGS_MODULE` de modo que mod_python conoce que configuración utilizar.

Nota que estamos usando la directiva `<Location>` y no `<Directory>`. Esta última se utiliza para apuntar a lugares de nuestra sistema de archivos, mientras que `<Location>` apunta a lugares en la estructura de la URL de un sitio web. `<Directory>` no tendría sentido aquí.

Apache comunmente corre como un usuario diferente de tu usuario normal y puede tener una ruta y un sys.path distintos. Puedes necesitar decirle a mod_python cómo encontrar tu proyecto y a Django mismo:

```
PythonPath "['/ruta/al/proyecto', '/ruta/a/django'] + sys.path"
```

También puedes agregar directivas como *PythonAutoReload Off*' para ajustar la performance. Mira la documentación de mod_python para un obtener un listado completo de opciones.

Ten en cuenta que deberias configurar `PythonDebug Off` en un servidor de producción. Si dejas `PythonDebug On`, tus usuarios verán feas trazas de error de Python si algo sale dentro de mod_python.

Reinicia Apache, y cualquier petición a tu sitio (o a tu host virtual si pusiste las directivas dentro de un bloque `<VirtualHost>`) será servida por Django.

Nota

Si implementas Django en un subdirectorio -- esto es, en algun lugar más profundo que "/" -- Django no recortará el prefijo de la URL para tu URLpatterns. Entonces, si tu configuración de Apache luce como esto:

```
<Location "/misitio/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE misitio.settings
    PythonDebug On
</Location>
```

entonces *todos* tus patrones de URL deberán comenzar con `"/misitio/"`. Por esta razón es que usualmente recomendamos implementar Django sobre la raiz de tu dominio o host virtual. Alternativamente, simplemente puede hacer descender el nivel de tu URL usando una cuña de URLconf:

```
urlpatterns = patterns('',
    (r'^misitio/', include('normal.root.urls')),
)
```

20.3.2. Corriendo multiples instalaciones de Django en la misma instancia Apache

Es enteramente posible correr multiples instalaciones de Django en la misma instancia de Apache. Probablemente quieras hacer esto si eres un desarrollador web independiente con multiples clientes pero un sólo un único servidor.

Para lograr esto, simplemente usa `VirtualHost` así:

```
NameVirtualHost *

<VirtualHost *>
    ServerName www.ejemplo.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE misitio.settings
</VirtualHost>

<VirtualHost *>
    ServerName www2.ejemplo.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE misitio.other_settings
</VirtualHost>
```

Si necesitar poner dos instalaciones de Django sobre el mismo `VirtualHost`, necesitar prestar especial atención para asegurarte de que el caché de código de mod_python no mezcle las cosas. Usa la directiva `PythonInterpreter` para brindar diferentes directivas `<Location>` a interpretes distintos:

```
<VirtualHost *>
    ServerName www.ejemplo.com
    # ...
    <Location "/algo">
        SetEnv DJANGO_SETTINGS_MODULE misitio.settings
        PythonInterpreter misitio
    </Location>

    <Location "/otracosa">
        SetEnv DJANGO_SETTINGS_MODULE misitio.other_settings
        PythonInterpreter misitio_otro
    </Location>
</VirtualHost>
```

Los valores de `PythonInterpreter` no importante realmente ya que se encuentran en dos bloques `Location` diferentes.

20.3.3. Corriendo un servidor de desarrollo con mod_python

Debido a que mod_python cachea el código python cargado, cuando implemantas sitios Django sobre mod_python necesitarás reiniciar Apache cada vez que realizar cambios en tu código. Esto puede ser tedioso, por lo que aqui compartimos un pequeño truco para evitarlo: simplemente agrega `MaxRequestsPerChild 1` a tu archivo de configuración para forzar a Apache a recargar todo con cada petición. Pero no hagas esto en un servidor de producción, o revocaremos tus privilegios Django.

Si eres el tipo de programador que depuran dispersando sentencias `print` por el código (nosotros somos), ten en cuenta que `print` no tiene efectos sobre mod_python; estas no aparecen en el log de Apache como pudrías esperar. Si necesitas imprimir información de depuración en una configuración mod_python, probablemente quieras usar el paquete de registro de eventos estándar de Python (Python's standard logging package). Hay más información disponible en http://docs.python.org/lib/module-logging.html. Alternativamente, puedes agregar la información de depuración a las plantillas de tu página.

20.3.4. Sirviendo Django y archivos multimedia desde la misma instancia Apache

Django no debería ser utilizado para servir archivos multimedia (imágen, audio, video, flash) por sí mismo; mejor deja ese trabajo al servidor web que hayas elegido. Recomendamos usar un servidor Web separado (es decir, uno que no está corriendo a la vez Django) para servir estos archivos. Para más información, mira la sección "Escalamiento".

Sin embargo, si no tienes opción para servir los archivos multimedia que no sea el mismo `VirtualHost` Apache que usa Django, aquí te mostramos como desactivar mod_python para una parte particular del sitio:

```
<Location "/media/">
    SetHandler None
</Location>
```

Cambia `Location` a la URL raiz donde se encuentran tus archivos.

Tambien puedes usar `<LocationMatch>` para comparar con una expresión regular. Por ejemplo, esto configura Django en la raiz del sitio pero deshabilitando Django para el subdirectorio `media` y cualquier URL que termine en `.jpg`, `.gif`, o `.png`:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "/media/">
    SetHandler None
</Location>
```

```
<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>
```

En todos estos casos, necesitarás configurar la directiva `DocumentRoot` para que Apache sepa dónde encontrar tus archivos estáticos.

20.3.5. Manejo de errores

Cuando usas Apache/mod_python, los errores serán canalizados por Django -- en otras palabras, estos no se propagan al nivel de Apache y no aparecerán en el `error_log` del servidor.

La excepción a esto sucede si algo está realmente desordenado en tu configuración Django. En este caso, verás una página "Internal Server Error" en tu navegador, y el volcado de error (traceback) de Python completo en tu archivo `error_log` de Apache. Este volcado de error se difunde por multiples líneas. (Sí, es feo y bastante difícil de leer, pero así como mod_python hace las cosas).

20.3.6. Manejando fallas de segmentación

Algunas veces, Apache produce fallas de segmentación (Segmentation faults, en inglés) cuando instalas Django. Cuando esto sucede, se trata casi *siempre* de una o dos causas no muy relacionadas con Django en sí:

- Puede ser que tu código Python está importando el módulo `pyexpat` (usado para parseo XML), lo que puede entrar en conflicto con la versión embebida en Apache. Para información detallada, revisa "Expat Causing Apache Crash" en http://www.djangoproject.com/r/articles/expat-apache-crash/.
- Puede deberse a que estás corriendo mod_python y mod_php sobre la misma instancia de Apache, con MySQL como motor de base de datos. En algunos casos, esto ocasiona un conocido problema que mod_python tiene debido a conflictos de versión en PHP y el back-end MySQL de la base. Hay información detallada en un listado FAQ de mod_python, accesible via http://www.djangoproject.com/r/articles/php-modpython-faq/

Si continuas teniendo problemas para configurar mod_python, una buena cosa para hacer es poner un esqueleto de sitio sobre mod_python a funcionar, sin el framework Django. Esta es una manera fácil de aislar los problemas específicos de mod_python. El artículo "Getting mod_python Working" detalla el procedimiento: http://www.djangoproject.com/r/articles/getting-modpython-working/.

El siguiente paso debería ser editar tu código de pruebas y agregar la importación de cualquier código específico de Django que estes usando -- tus vistas, tus modelos, tu URLconf, la configuración de RSS, y así. Incluye estas importaciones en tu función de gestión de pruebas, y accede a la URL correspondiente desde tu navegador. Si esto causa un colapso, habrás confirmado que es la importación de código Django la causa del problema. Gradulamente reduce el conjunto de importaciones hasta que el colapso desaparezca, de manera de encontrar el módulo específico que es el culpable. Profundiza en los módulos y revisa sus importaciones si es necesario. Para más ayuda, herramientas de sistema como `ldconfig` en Linux, `otool` en Mac OS, y `ListDLLs` (de SysInternals) en Windows pueden ayudarte a indentificar dependencias compartidas y posibles conflictos de version.

20.4. Usando Django con FastCGI

Aunque Django bajo Apache y mod_python es la configuración más robusta de implementación, mucha gente usa hosting compartido, en los que FastCGI es la única opción de implementación.

Adicionalmente, en algunas situaciones, FastCGI permite mayor seguridad y posiblemente una mejor performance que mod_python. Para sitios pequeños, FastCGI además puede ser más liviano que Apache.

20.4.1. Descripción de FastCGI

FastCGI es una manera eficiente de dejar que una aplicación externa genere páginas para un servidor Web. El servidor delega las peticiones Web entrantes (a través de un socket) a FastCGI, quien ejecuta el código y devuelve la respuesta al servidor, quien, a su turno, la remitirá al navegador del cliente.

Como mod_python, FastCGI permite que el código permanezca en memoria, logrando que las peticiones sean servidas sin tiempo de inicialización. A diferencia de mod_python, un proceso FastCGI no corre dentro del proceso del servidor Web, sino en un proceso separado y persistente.

> **¿Por qué ejecutar código en un proceso separado?**
> Los módulos tradicionales `mod_*` en Apache embeben varios lenguajes de scripting (los más notables son PHP, Python/mod_python, y Perl/mod_perl) dentro del espacion de procesos de tu servidor Web. A pesar de que esto reduce el tiempo de inicio (porque el código no tiene que ser leído del disco para cara consulta), lo hace al costo de ocupar espacio en memoria.
> Cada proceso Apache consigue una copia completa del motor de Apache, con todas las características de Apache que Django simplemente no aprovecha. Los procesos FastCGI, por otro lado, solo tienen el overhead de memoria de Python y Django.
> Debido a la naturaleza de FastCGI, también es posible tener procesos ejecutando bajo una cuenta de usuario diferente de la del proceso del servidor Web. Este es un buen beneficio de seguridad es sistemas compartidos, dado que significa que puedes asegurar tu código de otros usuarios.

Antes de que puedas empezar a usar FastCGI con Django, necesitas instalar `flup`, una biblioteca Python para manejar FastCGI. Algunos usuarios han reportado páginas que explotaron con versiones antiguas de `flup`, por lo cual puedes querer utilizar la última versión SVN. Puedes conseguir `flup` en http://www.djangoproject.com/r/flup/.

20.4.2. Ejecutando tu Servidor FastCGI

FastCGI opera sobre un modelo cliente/servidor, y en la mayoría de los casos estarás iniciando el proceso servidor FastCGI por tu cuenta. Tu servidor Web (ya sea Apache, lighttpd, o algún otro) hace contacto con tu proceso Django-FastCGI solo cuando el servidor necesita cargar una página dinámica. Como el demonio ya está ejecutando su código en memoria, puede servir la respuesta muy rápido.

> **Nota**
> Si estás en un sistema de hosting compartido, probablemente estés forzado a usar procesos FastCGI manejados por el Web server. Si estás en esta situación, debes leer la sección titulada "Ejecutando Django en un proveedor de Hosting compartido con Apache", más abajo.

Un servidor Web puede conectarse a un servidor FastCGI de dos formas: usando un socket de dominio Unix, (un *named pipe* en sistemas Win32) o un socket TCP. Lo que elijas es una cuestión de preferencias; usualmente un socket TCP es más fácil debido a cuestiones de permisos.

Para iniciar tu servidor, primero cambia al directorio de tu proyecto (donde está tu `manage.py`), y ejecuta `manage.py` con el comando `runfcgi`:

```
./manage.py runfcgi [options]
```

Si especificas `help` como única opción después de `runfcgi`, se mostrará una lista de todas las opciones disponibles.

Necesitarás especificar un `socket` o si no `host` y `port`. Entonces, cuando configures tu servidor Web, solo necesitas apuntarlo al socket o host/port que especificaste cuando iniciaste el servidor FastCGI.

Algunos ejemplos pueden ayudar a explicarlo:

- Ejecutar un servidor 'threaded' en un puerto TCP:

  ```
  ./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
  ```

- Ejecutar un servidor preforked sobre un socket de dominio Unix:

  ```
  ./manage.py runfcgi method=prefork socket=/home/user/mysite.sock pidfile=django.pid
  ```

- Ejecutar sin demonizar (ejecutar en segundo plano) el proceso (es bueno para el debugging):

  ```
  ./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock
  ```

Detener el Demonio FastCGI

Si tienes el proceso ejecutando en primer plano, es fácil detenerlo: simplemente presiona Ctrl+C para detenerlo y salir del servidor FastCGI. Si estás tratando con procesos en segundo plano, necesitarás recurrir al comando `kill` de Unix.

Si especificas la opción `pidfile` en `manage.py runfcgi`, puedes detener el demonio FastCGI en ejecución de esta forma:

```
kill `cat $PIDFILE`
```

donde `$PIDFILE` es el `pidfile` que especificaste.

Para reiniciar con facilidad tu demonio FastCGI en Unix, pedes usar este breve script en la línea de comandos:

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env - \
  PYTHONPATH="../python:.." \
  ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

20.4.3. Usando Django con Apache y FastCGI

Para usar Django con Apache y FastCGI, necesitarás que Apache esté instalado y configurado, con mod_fastcgi instalado y habilitado. Consulta la documentación de Apache y mod_fastcgi para instrucciones detalladas: http://www.djangoproject.c

Una vez que hayas completado la configuración, apunta Apache a tu instancia FastCGI de Django editando el archivo `httpd.conf` (de la configuración de Apache). Necesitarás hacer dos cosas:

- Usar la directiva `FastCGIExternalServer` para especificar la localización de tu servidor FastCGI.
- Usar `mod_rewrite` para apuntar las URLs a FastCGI según sea necesario.

Especificando la Localización del Servidor FastCGI

La directiva `FastCGIExternalServer` le dice a Apache como encontrar tu servidor FastCGI. Como se explica en los documentos de FastCGIExternalServer (http://www.djangoproject.com/r/mod_fastcgi/FastCGIExternalServer/), puedes especificar un `socket` o un `host`. Aquí hay ejemplos de ambos:

```
# Connect to FastCGI via a socket/named pipe:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

En los dos casos, el directorio /home/user/public_html/ debe existir, aunque el archivo `/home/user/public_html/mysite.fcgi` no necesariamente tiene que existir. Es solo una URL usada por el servidor Web internamente -- un enganche para indicar que las consultas en esa URL deben ser manejadas por FastCGI. (Más sobre esto en la siguiente sección.)

Usando mod_rewrite para apuntar URLs hacia FastCGI

El segundo paso es decirle a Apache que use FastCGI para las URLS que coincidan con cierto patrón. PArа hacer esto, usa el módulo mod_rewrite y reescribe las URLs hacia `mysite.fcgi` (o donde hayas especificado en la directiva `FastCGIExternalServer`, como se explicó en la sección anterior).

En este ejemplo, le decimos a Apache que use FastCGI para manejar cualquier consulta que no represente un archivo del sistema de archivos y no empiece con `/media/`. Probablemente éste sea el caso más común, si estás usando el sitio de administración de Django:

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L]
```

```
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
  </VirtualHost>
```

20.4.4. FastCGI y lighttpd

lighttpd (http://www.djangoproject.com/r/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 modulos, en algún lugar después de `mod_rewrite` y `mod_access`, y antes de `mod_accesslog`. Probablemente desees también `mod_alias`, para servir medios de administración.

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.

20.4.5. Ejecutando Django en un Proveedor de Hosting Compartido con Apache

Muchos proveedores de hosting compartido no te permiten ejecutar tus propios demonios servidores o editar el archivo `httpd.conf`. En estos casos, aún es posible ejecutar Django usando procesos iniciados por el sevidor Web.

> **Nota**
> Si estás usando procesos iniciados por el servidor Web, como se explica en esta sección, no necesitas iniciar el servidor FastCGI por tu cuenta. Apache iniciará una cantidad de procesos, escalando según lo necesite.

En el directorio raíz de tu Web, agrega esto a un archivo llamado `.htaccess`

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

Después, crea un pequeño script que le diga a Apache como iniciar tu programa FastCGI. Crea un archivo `mysite.fcgi`, colócalo en tu directorio Web, y asegúrate de hacerlo ejecutable:

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

Reiniciando el Server Iniciado

Si cambias cualquier código Python en tu sitio, necesitarás decirle a FastCGI que el código ha cambiado. Pero no hay necesidad de reiniciar Apache en este caso the. Sólo volver a subir `mysite.fcgi` -- o editar el archivo -- de manera que la fecha y hora del archivo cambien. Cuando Apache ve que el archivo ha sido actualizado, reiniciará tu aplicación Django por ti.

Si tienen acceso a la línea de comandos en un sistema Unix system, puedes hacer esto fácilmente usando el comando `touch`:

```
touch mysite.fcgi
```

20.5. Escalamiento

Ahora que sabes como tener a Django ejecutando en un servidor simple, veamos como puedes escalar una instalación Django. Esta sección explica como puede escalar un sitio desde un servidor único a un cluster de gran escala que pueda servir millones de hits 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 mostrat el principio general, y cuando sea posible, trateremos de señalar donde se puedan elegir distintas opciones.

Primero, haremos una buena presuposición, y hablaremos exclusivamente acerca de escalamiento bajo Apache y mod_python. A pesar de que conocemos vario casos exitosos de desarrollos FastCGI medios y grandes, estamos mucho más familiarizados con Apache.

20.5.1. Ejecutando en un Servidor Único

LA mayoría de los sitios empiezan ejecutando en un servidor único, con una arquitectura que se ve como en la Figura 20-1.

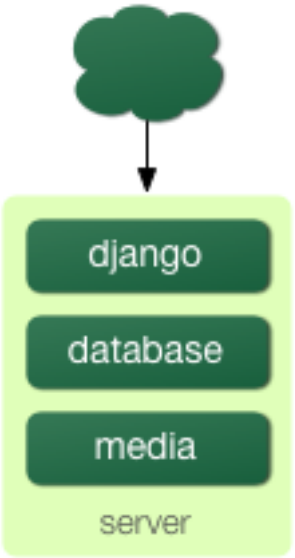

Figura 20.1: configuración de Django en un servidor único .

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.

Sin embargo, a medida que el tráfico se incemente, 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 servidos 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.

20.5.2. 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 Figura 20-2.

Aqui es donde empezamos a movernos hacia lo que usualmente se llama arquitectura *n-tier*. No te asustes por la terminilogía -- 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.

20.5.3. Ejecutando un Servidor de Medios Separado

Aún tenemos un gran problema por delante desde la configuración del servidor único: el servicio de medios 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 particularmente buena performance. 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 Figura 20-3).

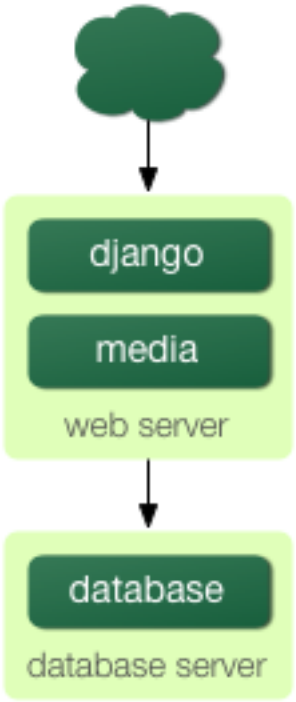

Figura 20.2: Moviendo la base de datos a un servidor dedicado.

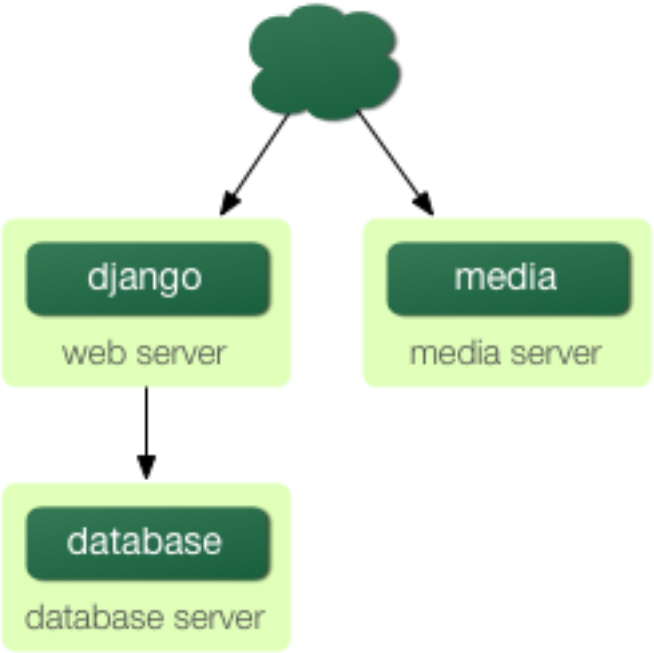

Figura 20.3: Separando el servidor de medios.

Idealmente, este servidor de medios debería correr un servidor Web desnudo, 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 'pelado' 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. El administrador de Django necesita poder escribir medios 'subidos' en el servidor de medios. (la configuración de `MEDIA_ROOT` controla donde se esciben estos medios). Si un medio habita en otro servidor, de todas formas necesitas organizar una forma de que esa escritura se pueda hacer a traves de la red.

La manera más fácil de hacer esto es usar el NFS para montar los directorios de medios del servidor de medios en el servidor Web (o los servidores Web). Si los montas en la misma ubicación apuntada por `MEDIA_ROOT`, el uploading de medios Simplemente Funciona¿.

20.5.4. Implementando Balance de Carga y Redundancia

A esta altura, ya hemos separado las cosas todo lo posible. Esta configuración de tres servers 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.

Actualmente, esto es algo bueno. Una mirada a la Figura 20-3 te permitirá cer que si falla aunque sea uno solo de los servidores, el sitio entero se cae. Asi que a medida que agregas servidores redundantes, no sólo incrementas capacidad, sino también confiabilidad.

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 Djando ejecutando en diferente hardware. -- simplemente copia el código en varias máquinas, y 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 caros y propietarios, 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` (ver Capítulo 13).

> **Nota**
>
> Si estás usando FastCGI, puedes realizar este mismo paso distribución y balance de carga separando los servidores Web front-end y los procesos FastCGI back-end en diferentes máquinas. El servidor front-end se convierte esencialmente en el balanceador de carga, y los procesos FastCGI back-end reemplaza a los servidores Apache/mod_python/Django.

Con los servidores Web en cluster, nuestra arquitectura en evolución empieza a verse más compleja, como se ve en la Figura 20-4.

Observar que en el diagrama nos referimos a los servidores Web como "el cluster" para indicar que el numero de servidores basicamente es variable. Una vez que tienes un balanceador de carga en el frente, pueder agregar y eliminar servidores Web back-end sin perder un segundo fuera de servicio.

20.5.5. Yendo a lo grande

En este punto, los siguientes pasos son derivaciones del último:

- A menida que necesites más performance en la base de datos, necesitarás agregar servidores de base de datos relicados. 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.

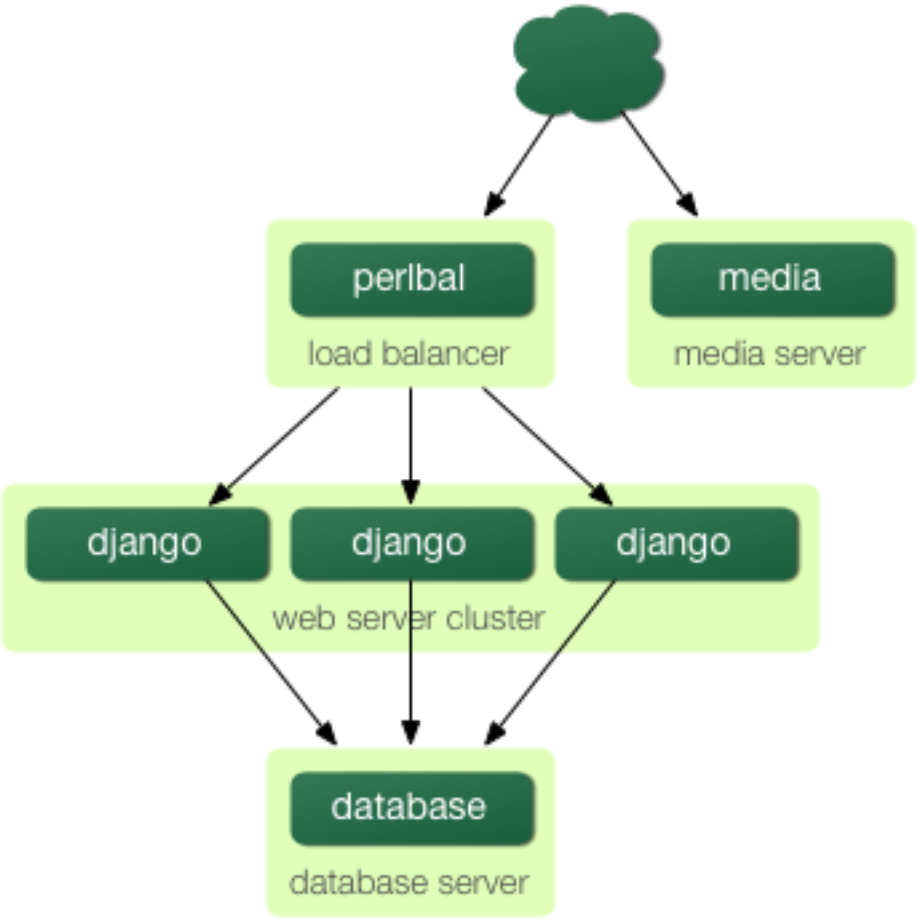

Figura 20.4: Configuración de un server redundante con balance de carga.

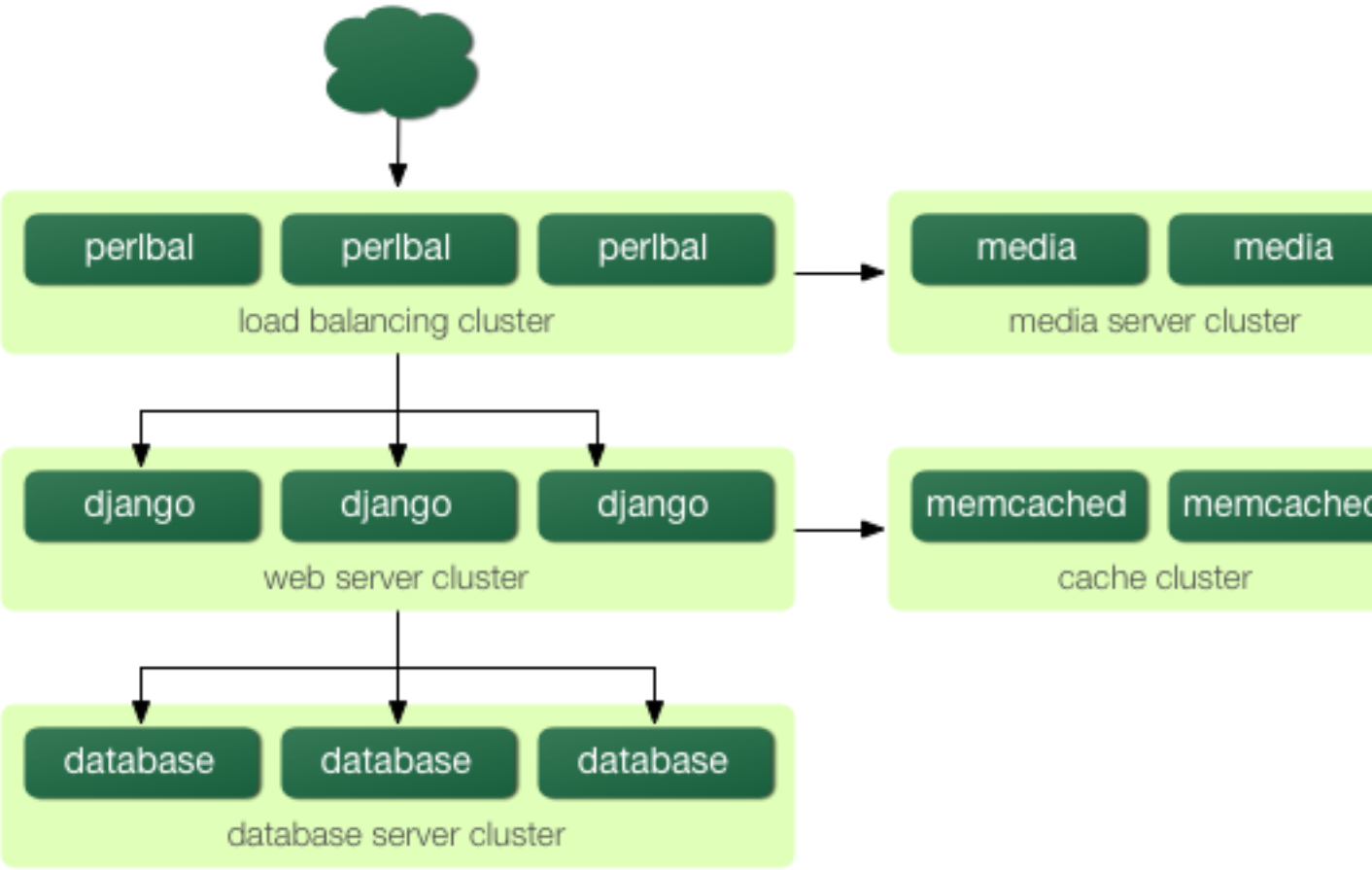

Figura 20.5: Un ejemplo de configuración de Django de gran escala.

Después de algunas de estas iteraciones, una arquitectura de gran escala debe verse como en la Figura 20-5.

A pesar de que mostramos solo dos o tres servidores en cada nivel, no hay un límite fundamental a cuantos puedes agregar.

Una vez que haz llegado a este nivel, te quedan pocas opciones. El Apéndice A tiene alguna información proveniente de desarrolladores responsables de algunas instalaciones Django de gran escala. Si estás planificando un sitio Django de alto tráfico, es una lectura recomendada.

20.6. 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 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 dificil de escribir sobre eso que sobre escalamiento. Si estás pensando seriamente en desplegar una aplicación Django de gran escala, deberás pasar un 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 descubiero a traves de los años.

20.6.1. No hay tal cosa como demasiada RAM

Cuando escribimos esto, la RAM realmente cara cuesta aproximadamente 200 dólares por gigabyte -- moneditas comparado con el tiempo empleado en ajustes de performance. 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 cuanto 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. La base de datos de LJWorld.com -- que incluye medio millón de artículos desde 1989 -- tiene menos de 2 GB.

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.

20.6.2. Deshabilita Keep-Alive

`Keep-Alive` es una característica de HTTP que permite que múltiples pedidos HTTP sean servidos sobre una conexión TCP única, evitando el overhead 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.

20.6.3. 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 -- ve directamente a memcached.

20.6.4. Usa memcached siempre

Por supuesto, seleccionar memcached no te hace mejor si no lo usas realmente. El Capítulo 13 es tu mejor amigo aquí: aprende como usar el framework de cache de Django, y usalo en todas partes que te sea posible. Un uso de cache agresivo y preemptico es usualmente lo único que se puede hacer para mantener un sitio funcionando bajo el mayor tráfico.

20.6.5. 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.

20.7. ¿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 tí o el próximo Google.

Apéndice A

Casos de estudio

Para ayudar a responder las preguntas acerca de cómo funciona Django en el "mundo real", hablamos (bueno, nos escribimos) con un puñado de personas que tienen sitios completos y funcionales desarrollados en Django. La mayor parte de este apéndice son sus palabras, que han sido ligeramente editadas para mayor claridad.

A.1. Elenco

Conozcamos a nuestros personajes y sus proyectos.

- *Ned Batchelder* es el ingeniero principal de Tabblo.com. Tabblo comenzó su vida como una herramienta para contar historias basadas en fotos compartidas, pero recientemente fue comprado por Hewlett-Packard para propósitos de mayor alcance:

  > HP vio valor real en nuestro estilo de desarrollo web, y en la forma en que tendimos puentes entre el mundo virtual y el mundo físico. Nos adquirieron de modo que pudiéramos llevar esa tecnología a otros sitios en la Web. Tabblo.com todavía es un gran sitio para contar historias, pero ahora también estamos trabajando en separar componentes y reutilizar las piezas más importantes de nuestra tecnología.

- *Johannes Beigel* es uno de los principales desarrolladores en Brainbot Technologies AG. El sitio Django más orientado al público de Brainbot es `http://pediapress.com/`, donde se pueden solicitar versiones impresas de artículos de Wikipedia. El equipo de Johannes actualmente está trabajando en un programa de gestión de conocimiento para empresas conocido como Brainfiler.

  Johannes nos cuenta que Brainfiler

  > [...] es una solución de software para gestionar, buscar, categorizar y compartir información de fuentes distribuidas. Está construido para el uso empresarial tanto en entornos de intranet como internet y es altamente escalable y personalizable. El desarrollo de los conceptos y componentes medulares comenzó en el 2001. Recientemente hemos rediseñado/reimplementado el servidor y el cliente web, que [ahora] está basado en Django.

- *David Cramer* es el desarrollador principal de Curse, Inc. Él desarrolló Curse.com, un sitio para devotos a los videojuegos multijugador masivos como World of Warcraft, Ultima Online, y otros.

  Curse.com es uno de los sitios sobre Django más grandes de Internet:

  > Tenemos aproximadamente entre 60 y 90 millones de visitas a la página en promedio por mes,y tuvimos picos de más de 130 millones de visitas (en un mes) usando Django. Somos un sitio web muy dinámico y orientado al usuario específicamente enfocado en juegos multijugador masivos, y uno de los sitios web más grandes acerca de World of Warcraft globalmente. Nuestro sitio web se estableció a principios de 2005, y desde finales de 2006 nos hemos estado expandiendo a juegos más allá de World of Warcraft.

- *Christian Hammond* es un ingeniero senior en VMware (un desarrollador líder de software de virtualización). También es el desarrollador principal de Review Board (`http://www.review-board.org/`), un sistema de revisión de código basado en Web. Review Board nació como un proyecto interno de VMware, pero ahora es open source:

  > A finales de 2006, David Trowbridge y yo discutíamos el proceso que usabamos en VMware para manejar la revisión de código. Antes de que la gente enviara código al repositorio, se

suponía que enviaran un diff de los cambios a una lista de correo para ser revisado. Todo se manejaba por email, y por supuesto, se volvió difícil seguir las revisiones que necesitaban mayor atención. Fue entonces que comenzamos a discutir potenciales soluciones para este problema.
En vez de escribir mis ideas, las puse en código. En poco tiempo, nació Review Board. Review Board ayuda a los desarrolladores, contribuidores y revisores a seguir la evolución del código propuesto a revisión y a mejorar la comunicación entre unos y otros. En vez de hacer una vaga referencia a una parte del código en un email, el revisor puede comentar directamente sobre el código. El código, junto a los comentarios, aparecerá entonces en las revisiones, dando a los desarrolladores suficiente contexto para trabajar más rápidamente en los cambios que hagan falta.
Review Board creció rápidamente en VMware. De hecho, mucho más rápido de lo esperado. En unas pocas semanas, tuvimos diez equipos usando Review Board. Sin embargo, este proyecto no es interno para VMware. Se decidió desde el día uno que debería ser open source y estar disponible para ser usado por cualquier compañía o proyecto.
Hicimos un anuncio de open source y habilitamos el sitio conjuntamente, que está disponible en http://www.review-board.org/. La respuesta a nuestro anuncio público fue tan impresionante como nuestro anuncio interno a VMware. En poco tiempo, nuestro servidor de demostración recibió más de 600 usuarios, y la gente comenzó a contribuir al proyecto en sí.
Review Board no es la única herramienta de revisión de código del mercado, pero es la primera que hemos visto que es abierta y tiene el extenso conjunto de características en el que hemos trabajado para que incluya. Esperamos que esto repercuta en beneficios de tiempo para muchos proyectos (open source o no).

A.2. ¿Por qué Django?

Le preguntamos a cada desarrollador por qué decidió usar Django, qué otras opciones fueron consideradas, y cómo se tomó la decisión definitiva de usar Django.

Ned Batchelder:

> Antes de unirme a Tabblo, Antonio Rodriguez (fundador/CTO de Tabblo) hizo una evaluación de Rails y Django, y encontró que ambos proveían un gran ambiente de desarrollo rápido *quick-out-of-the-blocks*. Comparando ambos, encontró que Django tenía una mayor profundidad técnica que haría más fácil construir un sitio robusto y escalable. También, el que Django esté basado en Python significó que tendríamos toda la riqueza del ecosistema Python para soportar nuestro trabajo. Esto definitivamente quedó demostrado a medida que construimos Tabblo.

Johannes Beigel:

> Como hemos estado codificando en Python por muchos años ya, y rápidamente comenzamos a usar el framework Twisted, Nevow era la solución más *natural* para resolver nuestras aplicaciones web. Pero pronto nos dimos cuenta que -- a pesar de la integración perfecta con Twisted -- muchas cosas eran un poco incómodas de realizar dentro de nuestros procesos de desarrollo ágil.
>
> Luego de investigar un poco en Internet, nos quedó claro que Django era el framework de desarrollo web más prometedor para nuestros requerimientos.
>
> Lo que nos condujo a Django fue la sintaxis de sus templates, pero pronto apreciamos todas las otras características que estaban incluidas, y así fue que "compramos" rápidamente a Django.
>
> Después de algunos años de desarrollo e implementación en paralelo de sistemas (Nevow todavía está en uso para algunos proyectos en sitios de clientes), llegamos a la conclusión de que Django es mucho menos incómodo, resultando en un código mucho mejor para mantener, y que es mucho más divertido para trabajar.

David Cramer:

> Escuché sobre Django en el verano de 2006, momento en que estabamos listos para hacer un reacondicionamiento de Curse, y decidimos investigarlo. Quedamos muy impresionado de lo que podía hacer, y todos los aspectos donde podía ahorrarnos tiempo. Lo conversamos y al decidirnos por Django comenzamos a escribir la tercera revisio del sitio web casi inmediatamente.

Christian Hammond:

> Había jugado con Django en un par de pequeños proyectos y quedé muy impresionado con él. Está basado en Python, del que soy un gran fanático, y esto lo hace fácil no sólo para desarrollar sitios y aplicaciones web, sino que también mantiene el trabajo organizado y mantenible. Esto siempre es un problema en PHP y Perl. Basado en experiencias del pasado, no necesité pensar mucho para meterme con Django.

A.3. Comenzando

Como Django es una herramienta relativamente nueva, no hay muchos desarrolladores experimentados ahí afuera que lo dominen ampliamente. Le preguntamos a nuestro "panel" cómo consiguieron que su equipo adquiriese velocidad con Django y qué consejos querrían compartir con nuevos desarrolladores Django.

Johannes Beigel:

> Luego de programar principalmente en C++ y Perl, nos cambiamos a Python y continuamos usando C++ para el código computacionalmente intensivo.
>
> [Aprendimos Django mientras] trabajamos con el tutorial, navegando la documentación para obtener una idea de lo que es posible (es fácil perderse muchas característica si sólo se sigue el tutorial), e intentando comprender los conceptos básicos detrás del middleware, objetos request, modelos de base de datos, etiquetas de plantillas, filtros personalizados, los formularios, autorización, localización... Luego podríamos revisar más profundamente esos tópicos cuando realmente los necesitáramos.

David Cramer:

> La documentación del sitio web es grandiosa. Pégate a ella.

Christian Hammond:

> David y yo teníamos una experiencia previa con Django, aunque era limitada. Hemos aprendido mucho mientras desarrollábamos Review Board. Le aconsejaría a los nuevos usuarios que lean la tan bien escrita documentación de Django y el libro que ahora están leyendo, ya que ambos han sido inestimables para nosotros.

No tuvimos que sobornar a Christian para conseguir esa declaración -- ¡lo juramos!

A.4. Portando código existente

Aunque Review Board y Tabblo fueron desarrollos desde cero, los otros sitios fueron portados desde código ya existente. Estábamos interesados en escuchar fue ese proceso.

Johannes Beigel:

> Comenzamos a "migrar" el sitio desde Nevow, pero pronto nos dimos cuenta de que queríamos cambiar tantos aspectos conceptuales (tanto en la interfaz de usuario como en la parte del servidor de aplicación) que empezamos todo de nuevo usando el código que teníamos meramente como una referencia.

David Cramer:

> El sitio anterior estaba escrito en PHP. Ir de PHP a Python fue grandioso programáticamente. El único detalle es que hay que tener mucho más cuidado con la gestión de memoria [ya que los procesos Django permanecen mucho más tiempo que los procesos PHP (que son simples ciclos)].

A.5. ¿Cómo les fue?

Ahora la pregunta del millón: ¿Cómo los trató Django? Estabamos especialmente interesados en escuchar dónde Django perdió fuerza -- es importante conocer en que aspectos tus armas son débiles *antes* de usarlas en la barricada.

Ned Batchelder:

> Django realmente nos permitió experimentar con las funcionalidades de nuestro sitio web. Tanto antes como una startup en busca del calor de clientes y negocios, como ahora como parte de HP y trabajando con un número de socios, tuvimos que ser muy ágiles cuando hubo que adaptar el software a nuevas demandas.

La separación de la funcionalidad en modelos, vistas y controladores nos brindó una modularidad que permitió elegir apropiadamente dónde extender y modificar. El ambiente Python de trasfondo nos dio la oportunidad de utilizar bibliotecas existentes para resolver problemas sin reinventar la rueda. PIL, PDFlib, ZSI, JSmin, y BeautifulSoup son sólo un puñado de bibliotecas que agregamos para hacer algunas tareas que eran engorrosas para nosotros.

La parte más difícil del nuestro uso de Django ha sido la relación entre los objetos de memoria y lo objetos de la base de datos, de algunas maneras. Primero, el Mapeo Objeto-relacional (ORM) de Django no asegura que dos referencias a la misma entrada en la base de datos sean el mismo objeto Python, por lo que puedes verte en situaciones donde dos partes del código intentan modificar la misma entrada y una de las copias está anticuada. Segundo, el modelo de desarrollo de Django te anima a basar tus modelos de objetos de datos en objetos de base de datos. A lo largo del tiempo hemos encontrado más y más usos de objetos de datos que no se ajustan a la base de datos, y tenemos que migrarlos desde su naturaleza asumiendo que su información se almacenará en una base.

Para una base de código grande y que se utilizará por mucho tiempo, definitivamente tiene sentido gastar tiempo estudiando las formas en que tus datos serán almacenados y accedidos, y construyendo alguna infraestructura que soporte esas formas.

También hemos agregado nuestra propia facilidad para la migración de la base por lo que los desarrolladores no tienen que aplicar parches SQL para mantener los actuales esquemas de base de datos funcionando. Los desarrolladores que cambian el esquema escriben una función Python para actualizar la base, y eso se aplica automáticamente cuando el servidor se inicia.

Johannes Beigel:

Consideramos Django como una plataforma muy satisfactoria que encaja perfectamente con la manera Pythonica de pensar. Casi todo simplemente funciona según lo previsto.

Una cosa que necesitó un poco de trabajo en nuestro proyecto en curso fue ajustar la configuración del archivo `settings.py` y la estructura de directorios/configuración (para aplicaciones, plantilla, datos locales, etc), porque implementamos un sistema altamente modular y configurable, donde todas las vistas de Django son métodos de algunas instancias de clase. Pero con la omnipotencia del dinámico código Python, fue posible hacerlo.

David Cramer:

Gestionamos la implementación de grandes aplicaciones de base de datos en un fin de semana. Esto nos hubiera llevado una o dos semanas hacerlo en el sitio web previo, en PHP. Django ha brillado exactamente donde queríamos que lo haga.

Ahora, aunque Django es una gran plataforma, es evidente que no está construido para las necesidades especificas que cualquiera necesite. Al tiempo del lanzamiento inicial del sitio web sobre Django, tuvimos nuestro mayor tráfico mensual del año, y no podíamos continuar. Durante los meses siguientes ajustamos algunos detalles, mayormente el hardware y el software que servía las peticiones a Django. Esto incluyó modificaciones de nuestra configuración de hardware, optimización de Django, y perfeccionar el software servidor que usábamos, que en ese entonces era lighttpd y FastCGI.

En mayo de 2007, Blizzard (los creadores de World of Warcraft) lanzaron otro parche bastante grande, como el que habían lanzado en diciembre cuando nuestro sitio fue lanzado en Django. La primer cosa que pasó por nuestras cabezas fue, "hey, si soportamos el aluvión de diciembre, esto de ninguna manera puede ser tan grande, deberíamos estar bien". Perdimos cerca de 12 horas antes de que los servidores comenzaran a sentir el calor. La pregunta surgió de nuevo: ¿Realmente era Django la mejor solución para lo que nosotros queríamos lograr?

Gracias a todo el gran apoyo de la comunidad, y largas noches, pudimos implementar varios arreglos "en caliente" sobre el sitio durante esos días. Los cambios introducidos (que con suerte estarán incorporados a Django en el momento que este libro vea la luz) permiten que con completa tranquilidad, aquellos (no cualquiera) que tengan que lidiar con 300 peticiones web por segundo, puedan hacerlo, con Django.

Christian Hammond:

Django nos permitió construir Review Board bastante rápidamente forzándonos a estar organizados a través de la separación de URL, vistas y plantillas, y proveyéndonos útiles componentes listos para usar, como la aplicación de autenticación, cacheo, y la abstracción de base de datos. La mayor parte de esto funcionó realmente bien para nosotros.

Siendo un aplicación web dinámica, tuvimos que escribir un montón de código JavaScript. Esta es una área en la que Django no ha podido ayudarnos realmente mucho. El soporte que provee Django para el uso de plantillas, etiquetas, filtros y formularios son grandiosos, pero no son fácilmente utilizables desde código JavaScript. Hay veces que quisiéramos usar una plantilla en particular o un filtro, pero no hay manera de usarlo desde JavaScript. Personalmente me gustaría ver que se incorporen a Django algunas soluciones creativa para esto.

A.6. Estructura de Equipo

A menudo, los proyectos son exitosos gracias a sus equipos, y no a tecnología elegida. Consultamos a nuestro panel sobre sus equipos de trabajo, y qué herramientas y técnicas utilizan para permanecer en carrera.

Ned Batchelder:

Somos un ambiente de Startup Web bastante estándar: Trac/SVN, cinco desarrolladores. Tenemos un servidor de desarrollo, un servidor de producción, un script desarrollado ad hoc, y así.

Ah, y amamos Memcached.

Johannes Beigel:

Usamos Trac como nuestro bug tracker y wiki, y recientemente reemplazamos Subversion+SVK por Mercurial (un sistema de control de versiones distribuido escrito en Python que maneja la ramificación y fusión con encanto)

Pienso que tenemos un proceso de desarrollo muy ágil, pero no seguimos una metodología "rígida" como Extreme Programming (aunque tomamos prestadas muchas ideas de ahí). Somos más bien programadores pragmáticos.

Tenemos un sistema de construcción automatizada (personalizado pero basado en SCons), y pruebas unitarias para casi todo.

David Cramer:

Nuestro equipo consiste en cuatro desarrolladores web, todos trabajando en la misma oficina de modo que es bastante fácil comunicarse. Nos basamos en herramientas comunes como SVN y Trac.

Christian Hammond:

Review Board tiene actualmente dos desarrolladores principales (David Trowbridge y yo) y un par de contribuidores. Estamos hospedados en Google Code y usamos su repositorio Subversion, issue tracker, y wiki. De hecho, usamos Review Board para revisar nuestros cambios antes de incorporarlos. Hacemos pruebas en nuestras computadores locales, tanto manualmente como por pruebas de unidad. Nuestros usuarios en VMware que usan Review Board todos los días nos proveen de un montón de feedback útil y reportes de errores, que intentamos incorporar en el programa.

A.7. Implementación

Los desarrolladores de Django toman la facilidad de implementación y escalamiento muy seriamente, por lo que siempre estamos interesados en escuchar sobre ensayos y tribulaciones del mundo real.

Ned Batchelder:

Hemos usado cacheo tanto en la capa de consulta como de respuesta para agilizar los tiempos de respuesta. Tenemos una configuración clásica: un multiplexor, varios servidores de aplicación, un servidor de base de datos. Eso ha funcionado bien para nosotros, porque podemos usar cacheo sobre el servidor de aplicación para evitar el acceso a la base de datos, y luego agregar tantos servidores de aplicación como necesitemos para manejar la demanda.

Johannes Beigel:

Servidores Linux, preferentemente Debian, con muchos gigas de RAM. Lighttpd como servidor Web, Pound como front-end HTTPS y balanceador de carga si se necesita, y Memcached como sistema de caché. SQLite para pequeñas bases de datos, Postgres si los datos crecen mucho, y cosas altamente especializadas de base de datos para nuestros componentes de búsqueda y de gestión de conocimiento.

David Cramer:

> Nuestra estructura todavía está para debatirse... (pero es esto actualmente):
>
> Cuando un usuario solicita el sitio se lo envía a un cluster de servidores Squid usando Lighttpd. Allí, los servidores verifican si el usuario está registrado en el sistema. Si no lo está, se le sirve una página cacheada. Un usuario autenticado es reenviado a un cluster de servidores corriendo Apache2 con mod_python (cada uno con una gran cantidad de memoria), donde luego cada uno hace uso de en un sistema Memcached distribuido y un bestial servidor de base de datos MySQL. Los datos estáticos, como archivos grandes o videos, (actualmente) estan hospedados en un servidor corriendo una instalación mínima de Django usando lighttpd con fastcgi. Más adelante, migraremos todos esos datos a un servicio similar al S3 de Amazon.

Christian Hammond:

> Hay dos servidores de producción en este momento. Uno está en VMware y consiste en una maquina virtual Ubuntu corriendo en VMware ESX. Usamos MySQL para la base de datos, Memcached para nuestro back-end de cacheo, y actualmente Apache para el servidor Web. Tenemos varios servidores potentes que pueden escalarse cuando lo requiramos. También podemos mudar MySQL o Memcached a otra maquina virtual a medida que nuestra base de usuarios crece.
>
> El segundo servidor de producción es el de Review Board mismo. La configuración es casi idéntica al anterior, excepto que la maquina virtual se basa en VMware Server.

Block quote ends without a blank line; unexpected unindent.

Apéndice B

Referencia de la definición de modelos

El Capítulo 5 explica lo básico de la definición de modelos, y lo utilizamos en el resto del libro. Existe un enorme rango de opciones disponibles que no se han cubierto en otro lado. Este apéndice explica toda opción disponible en la definición de modelos.

A pesar de que estas APIs se consideran muy estables, los desarrolladores de Django agregan en forma consistente nuevos atajos y conveniencias a la definición de modelos. Es buena idea chequear siempre la última documentación online en http://www.djangoproject.com/documentation/0.96/model-api/. */0.96?/*

B.1. Campos

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 define.

> **Restricciones en el nombre de los campos**
>
> Django pone solo dos restricciones en el nombre de los campos:
>
> 1. Un nombre de campo no puede ser una palabra reservada de Python, porque eso ocasionaría un error de sintaxis en Python, por ejemplo:
>
>  ```
>  class Example(models.Model):
>  pass = models.IntegerField() # 'pass' es una palabra reservada!
>  ```
>
> 2. Un nombre de campo no puede contener dos o más guiones bajos consecutivos, debido a la forma en que trabaja la sintaxis de las consultas de búsqueda de , por ejemplo:
>
>  ```
>  class Example(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 "escapea" todos los nombres de tabla y columna de la base de datos en cada consulta SQL subyacente. Utiliza la sintaxis de "quoteo" 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.

B.1.1. 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.

B.1.2. BooleanField

Un campo Verdadero/Falso.

B.1.3. 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.

B.1.4. CommaSeparatedIntegerField

Un campo de enteros separados por comas. Igual que en `CharField`, se requiere el argumento `max_length`.

B.1.5. DateField

Un campo de fecha. `DateField` tiene algunos argumentos opcionales extra, como se muestra en la Tabla B-1.

Cuadro B.1: Argumentos opcionales extra de DateField

Argumento	Descripción
`auto_now`	Asigna automáticamente al campo un valor igual al momento en que se salva el objeto. Es útil para las marcas de tiempo "última modificación". Observar que *siempre* se usa la fecha actual; no es un valor por omisión que se pueda sobreescribir.
`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.

B.1.6. DateTimeField

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

B.1.7. EmailField

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

B.1.8. FileField

Un campo de upload de archivos. Tiene un argumento *requerido*, como se ve en la Tabla B-3.

Cuadro B.2: Opciones extra de FileField

Argumento	Descripción
`upload_to`	Una ruta del sistema de archivos local que se agregará a la configuración de `MEDIA_ROOT` para determinar el resultado de la función de ayuda `get_<fieldname>_url()`

Esta ruta puede contener formato `strftime` (ver http://www.djangoproject.com/r/python/strftime/), que será reemplazada por la fecha y hora del upload del archivo (de manera que los archivos subidos no llenen el directorio

dada).

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

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

Por ejemplo, digamos que tu `MEDIA_ROOT` es `'/home/media'`, y `upload_to` es `'photos/%Y/%m/%d'`. La parte `'%Y/%m/%d'` de `upload_to` es formato strftime; `'%Y'` es el año en cuatro dígitos, `'%m'` es el mes en dos 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 usar los métodos `get_FIELD_filename()`, `get_FIELD_url()`, y `get_FIELD_size()`. Ver el Apéndice C para una explicación completa de estos métodos.

> **Nota**
>
> Cualquiera sea la forma en que manejes tus archivos subidos, tienes que prestar mucha atención a donde los estás subiendo y que tipo de archivos son, para evitar huecos en la seguridad. *Valida todos los archivos subidos* para asegurarte que esos archivos son lo que piensas que son.
> Por ejemplo, si dejas que cualquiera suba archivos ciegamente, sin validación, a un directorio que está dentro de la raíz de documentos (*document root*) de tu servidor web, alguien podría subir un script CGI o PHP y ejecutarlo visitando su URL en tu sitio. ¡No permitas que pase!

B.1.9. 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 B-3.

Cuadro B.3: Opciones extra de FilePathField

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 matchear con un archivo llamado `foo23.txt`, pero no con `bar.txt` o `foo23.gif`).
`recursive`	Opcional; `True` o `False`. El valor por omisión es `False`. Especifica si deben incluirse todos los subdirectorios de `path`.

Por supuesto, estos argumentos pueden usarse juntos.

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

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

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

B.1.10. FloatField

Un numero de punto flotante, representado en Python por una instancia de `float`. Tiene dos argumentos requeridos, que se muestran en la Tabla B-4.

Cuadro B.4: Opciones extra de FloatField

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

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

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

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

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

B.1.11. ImageField

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

Además de los métodos especiales `get_FIELD_*` que están disponibles para `FileField`, un `ImageField` tiene también los métodos `get_FIELD_height()` y `get_FIELD_width()`. Éstos están documentados en el Apéndice C.

`ImageField` requiere la biblioteca Python Imaging Library (`http://www.pythonware.com/products/pil/`).

B.1.12. IntegerField

Un entero.

B.1.13. IPAddressField

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

B.1.14. NullBooleanField

Similar a `BooleanField`, pero permite `None`/`NULL` como opciones. Usar éste en lugar de un `BooleanField` con `null=True`.

B.1.15. PhoneNumberField

Un `CharField` que chequea que el valor es un teléfono válido estilo U.S. (en formato `XXX-XXX-XXXX`).

> **Nota**
>
> Si necesitas representar teléfonos de otros países, consulta el paquete `django.contrib.localflavor` para ver si ya están incluidas las definiciones para tu país.

B.1.16. PositiveIntegerField

Similar a `IntegerField`, pero debe ser positivo.

B.1.17. PositiveSmallIntegerField

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

B.1.18. SlugField

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

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

Un `SlugField` implica `db_index=True` debido a que son los se usan 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 autorellenar el slug, vía JavaScript, en el formulario de administración del objeto:

```
models.SlugField(prepopulate_from=("pre_name", "name"))
```

`prepopulate_from` no acepta nombres `DateTimeField` como argumentos.

B.1.19. SmallIntegerField

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

B.1.20. TextField

Un campo de texto de longitud ilimitada.

B.1.21. TimeField

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

B.1.22. URLField

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

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

B.1.23. USStateField

Una abreviatura de dos letras de un estado de U.S.

> **Nota**
>
> Si necesitas representar otros países o estados, busca en el paquete `django.contrib.localflavor` para ver si Django ya incluye los campos para tu localización.

B.1.24. XMLField

Un `TextField` que chequea que el valor sea un XML válido que matchea con un esquema dado. Tiene un argumento requerido, `schema_path`, que es la ruta en el sistema de archivos a un esquema RELAX NG (http://www.relaxng.org/) contra el cual validar el campo.

Requiere `jing` (http://thaiopensource.com/relaxng/jing.html) para validar el XML.

B.2. Opciones Universales de Campo

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

B.2.1. null

Si está en `True`, Django almacenará valores vacíos como `NULL` en la base de datos. El valor por omisión es `False`.

Observar 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`.

B.2.2. blank

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

Observar 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 `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.

B.2.3. 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 actual a ser almacenado. El segundo elemento es el nombre legible por humanos para la opción.

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

```
class Foo(models.Model):
    GENDER_CHOICES = (
        ('M', 'Male'),
        ('F', 'Female'),
    )
    gender = models.CharField(maxlength=1, choices=GENDER_CHOICES)
```

o fuera de la clase del modelo:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)
class Foo(models.Model):
    gender = models.CharField(maxlength=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. Ver Apéndice C para más detalles.

B.2.4. 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.

B.2.5. 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 syncdb`).

B.2.6. default

El valor por omisión del campo.

B.2.7. editable

Si es `False`, el campo no será editable en la interfaz de administración o via procesamiento de formularios. El valor por omisión es `True`.

B.2.8. 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.

B.2.9. primary_key

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

Su no se especifica `primary_key=True` para ningún campo del modelo, Django agregará automáticamente este campo:

```
id = 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.

B.2.10. radio_admin

Por omisión,el administrador de Django usa una interfaz de cuadro de selección (<select>) para campos que son `ForeignKey` o tienen `choices`. Si `radio_admin` es `True`, Django utilizará una interfaz radio-button en su lugar.

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

B.2.11. unique

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

B.2.12. 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 Story(models.Model):
    pub_date = models.DateTimeField()
    slug = models.SlugField(unique_for_date="pub_date")
    ...
```

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.

B.2.13. unique_for_month

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

B.2.14. unique_for_year

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

B.2.15. verbose_name

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

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

```
first_name = models.CharField("Person's first name", maxlength=30)
```

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

```
first_name = models.CharField(maxlength=30)
```

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

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

La convención es no capitalizar la primera letra del `verbose_name`. Django convertirá a mayúscula automáticamente la primera letra cuando lo necesite.

B.3. 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.

Sin embargo, la semántica de las relaciones uno-a-uno esta siendo revisada mientras se imprime este libro, así que no se cubren en esta sección. Consulte en la documentación on-line la información más actualizada.

B.3.1. Relaciones Muchos-a-Uno

Para definir una relación muchos-a-uno, usa `ForeignKey`. 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 `Car` tiene un `Manufacturer` -- es decir, un `Manufacturer` fabrica múltiples autos pero cada `Car` tiene solo un `Manufacturer` -- usa la siguiente definición:

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

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer)
    ...
```

Para crear una relación *recursiva* -- un objeto que tiene una relación muchos-a-uno con él mismo -- usa `models.ForeignKey('self`

```
class Employee(models.Model):
    manager = models.ForeignKey('self')
```

Si necesitas crear una relación con un modelo que aún no se ha definido, puedes usar el nombre del modelo en lugar del objeto modelo:

```
class Car(models.Model):
    manufacturer = models.ForeignKey('Manufacturer')
    ...

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

Observar que 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.

Detrás de la 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 `Car`, tendrá una columna `manufacturer_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` (`manufacturer` en el ejemplo) sea el nombre del modelo en minúsculas. Por supuesto, puedes ponerle el nombre que quieras. Por ejemplo:

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(Manufacturer)
    # ...
```

Los campos `ForeignKey` reciben algunos argumentos extra para definir como debe trabajar la relación (ver Tabla B-5). Todos son opcionales.

Cuadro B.5: Opciones de ForeignKey

Argumento	Descripción
`edit_inline`	Si no es `False`, este objeto relacionado se edita "inline" en la página del objeto relacionado. Esto significa que el objeto no tendrá su propia interfaz de administración. Usa `models.TABULAR` o `models.STACKED`, que designan si los objetos editables inline se muestran como una tabla o como una pila de conjuntos de campos, respectivamente.
`limit_choices_to`	Un diccionario para buscar argumentos y valores (ver el Apéndice C) que limita las opciones de administración disponibles para este objeto. Usa esto con funciones del módulo `datetime` de Python para limitar las opciones de fecha de los objetos. Por ejemplo: `limit_choices_to = {'pub_date__lte': datetime.now}` sólo permite la elección de objetos relacionados con `pub_date` anterior a la fecha/hora actual. En lugar de un diccionario, esto puede ser un objeto `Q` (ver Apéndice C) para consultas más complejas. No es compatible con `edit_inline`.
`max_num_in_admin`	Para objetos editados inline, este es el número máximo de objetos relacionados a mostrar en la interfaz de administración. Por lo tanto, si una pizza puede tener como máximo diez ingredientes, `max_num_in_admin=10` asegurará que un usuario nunca ingresará más de diez ingredientes. Observar que esto no asegura que no se puedan crear más de diez ingredientes relacionados. Simplemente controla la interfaz de administración; no fortalece cosas a nivel de Python API o base de datos.
`min_num_in_admin`	La cantidad mínima de objetos relacionados que se muestran en la interfaz de administración. Normalmente, en el momento de la creación se muestran `num_in_admin` objetos inline , y en el momento de edición se muestran `num_extra_on_change` objetos en blanco además de todos los objetos relacionados preexistentes. De todas formas, nunca se mostrarán menos de `min_num_in_admin` objetos relacionados.
`num_extra_on_change`	La cantidad de campos en blanco extra de objetos relacionados a mostrar en el momento de realizar cambios.

Cuadro B.5: Opciones de ForeignKey

Argumento	Descripción
`num_in_admin`	El valor por omisión de la cantidad de objetos inline a mostrar en la página del objeto en el momento de agregar.
`raw_id_admin`	Solo muestra un campo para ingresar un entero en lugar de un menú desplegable. Esto es útil cuando se relaciona con un tipo de objeto que tiene demasiadas filas para que sea práctico utilizar una caja de selección. No es utilizado con `edit_inline`.
`related_name`	El nombre a utilizar para la relación desde el objeto relacionado de hacia éste objeto. Para más información, ver el Apéndice C.
`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.

B.3.2. Relaciones Muchos-a-Muchos

Para definir una relación muchos-a-muchos, usa `ManyToManyField`. Al igual que `ForeignKey`, `ManyToManyField` requiere un argumento posicional: la clase a la cual se relaciona el modelo.

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

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

class Pizza(models.Model):
    toppings = models.ManyToManyField(Topping)
    ...
```

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` (`toppings`, 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 cual 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 `toppings` están en la `Pizza` (en lugar de que el `Topping` tenga `pizzas` `ManyToManyField`) porque es más natural pensar que una `Pizza` tiene varios ingredientes (toppings) que pensar que un ingrediente está en muchas pizzas. En la forma en que está configurado el ejemplo, el formulario de administración de la"Pizza" permitirá que los usuarios selecciones los ingredientes.

Los objetos `ManyToManyField` toman algunos argumentos extra para definir como debe trabajar la relación (ver Tabla B-6). Todos son opcionales.

Cuadro B.6: Opciones de ManyToManyField

Argumento	Descripción
`related_name`	El nombre a utilizar para la relación desde el objeto relacionado hacia este objeto. Ver Apéndice C para más información.
`filter_interface`	Usa una interfaz de "filtro" JavaScript agradable y discreta en lugar de la menos cómoda `<select multiple>` en el formulario administrativo de este objeto. El valos debe ser `models.HORIZONTAL` o `models.VERTICAL` (es decir, la interfaz debe apilarse horizontal o verticalmente).
`limit_choices_to`	Ver la descripción en `ForeignKey`.

Cuadro B.6: Opciones de ManyToManyField

Argumento	Descripción

`symmetrical`	Solo utilizado en la definición de `ManyToManyField` sobre sí mismo. Considera el siguiente modelo:
	`class Person(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 `person_set` a la clase `Person`. 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.

`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.

B.4. Opciones de los Metadatos del Modelo

Los metadatos específicos de un modelo viven en una `class Meta` definida en el cuerpo de tu clase modelo:

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

    class Meta:
        # model metadata options go here
        ...
```

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

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

B.4.1. 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 `books` (creada por `manage.py startapp books`), un modelo definido como `class Book` tendrá una tabla en la base de datos llamada `books`.

Para sobreescribir el nombre de la tabla de la base de datos, use el parámetro `db_table` dentro de `class Meta`:

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

    class Meta:
        db_table = 'things_to_read'
```

Si no se define, Django utilizará `app_label + '_' + model_class_name`. Ver la sección "**'Nombres de Tabla'_**" para más información.

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 quotea los nombres de tabla y de columna detrás de la escena.

B.4.2. 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:

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

    class Meta:
        get_latest_by = "order_date"
```

Ver el Apéndice C para más información sobre el método `latest()`.

B.4.3. order_with_respect_to

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

```
class Answer(models.Model):
    question = models.ForeignKey(Question)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

B.4.4. ordering

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

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

    class Meta:
        ordering = ['title']
```

Esto es una tupla o lista de 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.

Por ejemplo, para ordenar por un campo `title` en orden ascendente (A-Z), usa esto:

```
ordering = ['title']
```

Para ordenar por `title` en orden descendente (Z-A), usa esto:

```
ordering = ['-title']
```

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

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

Observar que no importa cuantos campos haya en `ordering`, el sitio de administración usa sólo el primer campo.

B.4.5. permissions

Permisos extra para almacenar en la tabla de permisos cuando se crea este objeto. Se crean automáticamente permisos para agregar, eliminar y cambiar para cada objeto que tenga establecida la opción `admin`. Este ejemplo especifica un permiso extra, `can_deliver_pizzas`:

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

    class Meta:
        permissions = (
            ("can_deliver_pizzas", "Can deliver pizzas"),
        )
```

Esto es una lista o tupla de dos tuplas de la forma `(permission_code, human_readable_permission_name)`. Ver el Capítulo 12 para más detalles sobre permisos.

B.4.6. unique_together

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

```
class Employee(models.Model):
    department = models.ForeignKey(Department)
    extension = models.CharField(maxlength=10)
    ...

    class Meta:
        unique_together = [("department", "extension")]
```

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`).

B.4.7. verbose_name

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

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

    class Meta:
        verbose_name = "order"
```

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

B.4.8. verbose_name_plural

El nombre del objeto en plural:

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

    class Meta:
        verbose_name_plural = "sphynges"
```

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

B.5. Managers

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

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

B.5.1. Nombres de Manager

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

```
from django.db import models

class Person(models.Model):
    ...

    people = models.Manager()
```

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

B.5.2. Managers Personalizados

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

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

Agregando Métodos Extra al Manager

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

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

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

```
from django.db import connection

class PollManager(models.Manager):

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

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

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

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

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

Modificando los QuerySets iniciales del Manager

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B.6. Métodos de Modelo

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

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

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

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

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

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

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

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

B.6.1. __str__

`__str__()` es un "método mágico" de Python que define lo que debe ser devuelto si llamas a `str()` sobre el objeto. Django usa `str(obj)` (o la función relacionada `unicode(obj)`, que se describe más abajo) en varios lugares, particularmente como el valor mostrado para hacer el render de un objeto en el sitio de administración de Django y como el valor insertado en un plantilla cuando muestra un objeto. Por eso, siempre debes retornar un string agradable y legible por humanos en el `__str__` de un objeto. A pesar de que esto no es requerido, es altamente recomendado.

Aquí hay un ejemplo:

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

```
    first_name = models.CharField(maxlength=50)
    last_name = models.CharField(maxlength=50)

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

B.6.2. get_absolute_url

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

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

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

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

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

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

Pero este es bueno:

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

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

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

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

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

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

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

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

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

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

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

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

B.6.3. Ejecutando SQL personalizado

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

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

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

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

B.6.4. Sobreescribiendo los Métodos por omisión del Modelo

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

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

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

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

También puedes evitar el guardado:

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

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

B.7. Opciones del Administrador

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

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

```
class Admin:
    pass
```

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

B.7.1. date_hierarchy

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

Aquí hay un ejemplo:

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

    class Admin:
        date_hierarchy = "order_date"
```

B.7.2. fields

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

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

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

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

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

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

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

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

fields

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

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

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

classes

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

Aplica múltiples clases separándolas con espacios:

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

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

description

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

B.7.3. js

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

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

B.7.4. list_display

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

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

Aquí hay algunos casos especiales a 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:

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

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

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

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

  Aquí está un modelo de ejemplo completo:

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

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

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

- Si el string dado es un método del modelo que retorna `True` o `False`, Django mostrará un bonito icono "on" o "off" si le das al método un atributo `boolean` con valor en `True`.

  Aquí está un modelo de ejemplo completo:

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

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

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

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

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

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

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

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

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

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

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

B.7.5. list_display_links

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

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

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

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

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

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

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

B.7.6. list_filter

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

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

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

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

B.7.7. list_per_page

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

B.7.8. list_select_related

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

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

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

B.7.9. ordering

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

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

B.7.10. save_as

Establece `save_as` a `True` para habilitar la característica "save as" en los formularios de cambios del administrador.

Normalmente, los objetos tienen tres opciones al guardar: "Save", "Save and continue editing" y "Save and add another". Si `save_as` es `True`, "Save and add another" será reemplazado por un botón "Save as".

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

Por omisión, `save_as` es `False`.

B.7.11. save_on_top

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

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

Por omisión, `save_on_top` es `False`.

B.7.12. search_fields

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

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

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

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

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

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

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

Cuadro B.7: Operadores Permitidos en search_fields

Operador	Significado

`^`	Matchea el principio del campo. Por ejemplo, si `search_fields` es `['^first_name', '^last_name']`, y un usuario busca `john lennon`, Django hará el equivalente a esta cláusula `WHERE` en SQL:
	`WHERE (first_name ILIKE 'john%' OR last_name ILIKE 'john%')`
	`AND (first_name ILIKE 'lennon%' OR last_name ILIKE 'lennon%')`
	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`.

`=`	Matchea exactamente, sin distinguir mayúsculas y minúsculas. Por ejemplo, si `search_fields` es `['=first_name', '=last_name']` y un usuario busca `john lennon`, Django hará el equivalente a esta clausula `WHERE` en SQL:
	`WHERE (first_name ILIKE 'john' OR last_name ILIKE 'john')`
	`AND (first_name ILIKE 'lennon' OR last_name ILIKE 'lennon')`
	Observar que la entrada de la consulta se divide por los espacios, por lo cual actualmente no es posible hacer una búsqueda de todos los registros en los cuales `first_name` 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.

Apéndice C

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 B. Una vez que hayas definido un modelo, usarás esta API en cualquier momento que necesites acceder a la base de datos. Has visto ejemplos del uso de esta API a través del libro; este apéndice explica todas las varias opciones detalladamente.

De manera similar a lo que ocurre con las APIs de modelos descriptos en el apéndice B, aunque estas APIs son considerados muy estables, 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/0.96/db-api/.

A lo largo de este apéndice, vamos a hacer referencia a los siguientes modelos, los cuales pueden formar una simple aplicación de blog:

```
from django.db import models

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

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()

    def __str__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateTimeField()
    authors = models.ManyToManyField(Author)

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

C.1. 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 mysite.blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> 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.

C.1.1. Qué pasa cuando grabas?

Cuando grabas un objeto, Django realiza los siguientes pasos:

1. **Emitir una señal pre_save.** Esto provee una notificación de que un objeto está a punto de ser grabado. Puedes registrar un ***listener*** que será invocado en cuanto esta señal sea emitida. Estas señales todavía están en desarrollo y no estaban documentadas cuando este libro fue a impresión; revisa la documentación en línea para obtener la información más actual.
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 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, `DateFields` 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. De nuevo, estas señales todavía no han sido documentadas.

C.1.2. Claves primarias autoincrementales

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(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id     # Returns None, because b doesn't have an ID yet.
None

>>> b2.save()
>>> b2.id     # Returns the ID of your new object.
14
```

No hay forma de saber cual será el valor de un identificador antes que llames a `save()` esto se debe a que ese valor es calculado por tu 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, name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> 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 `'Cheddar Talk'`, este ejemplo sobrescribiría el registro previo en la base de datos:

```
>>> b4 = Blog(id=3, name='Not Cheddar', tagline='Anything but cheese.')
>>> b4.save()  # Overrides the previous blog with 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.

C.2. 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` `b5` que ya ha sido grabada en la base de datos, este ejemplo cambia su nombre y actualiza su registro en la base:

```
>>> b5.name = 'New name'
>>> b5.save()
```

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()`.

Como 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:

```
>>> joe = Author.objects.create(name="Joe")
>>> entry.author = joe
>>> entry.save()
```

Django se quejará si intentas asignar un objeto del tipo incorrecto.

C.3. Recuperando objetos

A través del libro has visto cómo se recuperan objetos usando código como el siguiente:

```
>>> blogs = Blog.objects.filter(author__name__contains="Joe")
```

Hay bastantes partes móviles detrás de escena aquí: cuando recuperas objetos de la base de datos, estás construyendo realmente un `QuerySet` usando el `Manager` del modelo. Este `QuerySet` sabe como ejecutar SQL y retornar los objetos solicitados.

El Apéndice B 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(name='Foo', tagline='Bar')
>>> b.objects
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Manager isn't accessible via Blog instances.
```

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.

C.4. Caching y QuerySets

Cada `QuerySet` contiene un cache, para minimizar el acceso a la base de datos. Es importante entender como funciona, para escribir código mas eficiente.

En un `QuerySet` recién creado, el cache esta 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.headline for e in Entry.objects.all()]
print [e.pub_date for e in Entry.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 `Entry` durante el pequeñísimo período de tiempo entre ambas peticiones.

Para evitar este problema, simplemente graba el `QuerySet` y re-úsalo:

```
queryset = Poll.objects.all()
print [p.headline for p in queryset] # Evaluate the query set.
print [p.pub_date for p in queryset] # Reuse the cache from the evaluation.
```

C.5. Filtrando objetos

La manera mas simple de recuperar objetos de una tabla es conseguirlos todos. Para hacer esto, usa el método `all()` en un `Manager`:

```
>>> Entry.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()`:

```
>>> y2006 = Entry.objects.filter(pub_date__year=2006)
>>> not2006 = Entry.objects.exclude(pub_date__year=2006)
```

Tanto `filter()` como `exclude()` toman argumentos de *patrones de búsqueda*, los cuales se discutirán detalladamente en breve.

C.5.1. Encadenando filtros

El resultado de refinar un `QuerySet` es otro `QuerySet` así que es posible enlazar refinamientos, por ejemplo:

```
>>> qs = Entry.objects.filter(headline__startswith='What')
>>> qs = qs..exclude(pub_date__gte=datetime.datetime.now())
>>> qs = qs.filter(pub_date__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 el. Por ejemplo, el siguiente `QuerySet` no es evaluado hasta que se iterado sobre él en el bucle `for`:

  ```
  qs = Entry.objects.filter(pub_date__year=2006)
  qs = qs.filter(headline__icontains="bill")
  for e in qs:
      print e.headline
  ```

  Esto imprime todos los títulos desde el 2006 que contienen "bill" pero genera solo un acceso a la base de datos.

- *Imprimiéndolo*: 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:

  ```
  >>> entry_list = list(Entry.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 = Entry.objects.filter(headline__startswith="What")
q2 = q1.exclude(pub_date__gte=datetime.now())
q3 = q1.filter(pub_date__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 `pub_date` 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 `pub_date` es mayor que el día de hoy. El `QuerySet` inicial (`q1`) no es afectado por el proceso de refinamiento.

C.5.2. 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`):

```
>>> Entry.objects.all()[:5]
```

Esto retorna las entradas desde la sexta hasta la décima (`OFFSET 5 LIMIT 5`):

```
>>> Entry.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:

```
>>> Entry.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 `Entry` en la base de datos, después de ordenar las entradas alfabéticamente por título:

```
>>> Entry.objects.order_by('headline')[0]
```

y es equivalente a lo siguiente:

```
>>> Entry.objects.order_by('headline')[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.

C.5.3. 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 mas 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(*campos)

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 (ver Apéndice B). Puedes sobrescribir esto para una consulta particular usando el método `order_by()`:

```
>>> Entry.objects.filter(pub_date__year=2005).order_by('-pub_date', 'headline')
```

Este resultado será ordenado por `pub_date` de forma descendente, luego por `headline` de forma ascendente. El signo negativo en frente de `"-pub_date"` indica orden *descendiente*. Si el `-` esta ausente se asume un orden ascendente. Para ordenar aleatoriamente, usa `"?"`, así:

```
>>> Entry.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(*campos)

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 los nombre de los atributos de los objetos modelo:

```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
[Beatles Blog]

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]
```

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, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}],
>>> Blog.objects.values('id', 'name')
[{'id': 1, 'name': '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(campo, tipo, orden)

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:

```
>>> Entry.objects.dates('pub_date', 'year')
[datetime.datetime(2005, 1, 1)]

>>> Entry.objects.dates('pub_date', 'month')
[datetime.datetime(2005, 2, 1), datetime.datetime(2005, 3, 1)]

>>> Entry.objects.dates('pub_date', 'day')
[datetime.datetime(2005, 2, 20), datetime.datetime(2005, 3, 20)]

>>> Entry.objects.dates('pub_date', 'day', order='DESC')
[datetime.datetime(2005, 3, 20), datetime.datetime(2005, 2, 20)]

>>> Entry.objects.filter(headline__contains='Lennon').dates('pub_date', '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:

```
# Hits the database.
>>> e = Entry.objects.get(id=5)

# Hits the database again to get the related Blog object.
>>> b = e.blog
```

Esta es una búsqueda `select_related`:

```
# Hits the database.
>>> e = Entry.objects.select_related().get(id=5)

# Doesn't hit the database, because e.blog has been prepopulated
# in the previous query.
>>> b = e.blog
```

`select_related()` sigue claves foráneas tan lejos como le sea posible. Si tienes los siguientes modelos:

```
class City(models.Model):
    # ...

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(City)

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person)
```

entonces una llamada a `Book.objects.select_related().get(id=4)` colocará en el cache la `Person` relacionada *y* la `City` relacionada:

```
>>> b = Book.objects.select_related().get(id=4)
>>> p = b.author         # Doesn't hit the database.
>>> c = p.hometown       # Doesn't hit the database.

>>> b = Book.objects.get(id=4) # No select_related() in this example.
>>> p = b.author         # Hits the database.
>>> c = p.hometown       # Hits the database.
```

Notar que `select_related` no sigue claves foráneas que tienen `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:

```
>>> Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
```

Como resultado, cada objeto `Entry` tendrá en este caso un atributo adicional, `is_recent`, un booleano que representará si el atributo `pub_date` del entry 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 `entry_count`, un entero que indica la cantidad de objetos `Entry` asociados al blog:

```
>>> subq = 'SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id'
>>> Blog.objects.extra(select={'entry_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:

```
>>> Entry.objects.extra(where=['id IN (3, 4, 5, 20)'])
```

Los parámetros `select` y `where` antes descriptos 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:

```
>>> Entry.objects.extra(where=['headline=%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:

```
Entry.objects.extra(where=["headline='%s'"% name])
```

Este es un ejemplo de lo que es correcto:

```
Entry.objects.extra(where=['headline=%s'], params=[name])
```

C.5.4. 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:

```
>>> Entry.objects.get(id='foo') # levanta Entry.DoesNotExist
```

La excepción `DoesNotExist` hereda de `django.core.exceptions.ObjectDoesNotExist`, así que puedes protegerte de múltiples excepciones `DoesNotExist`:

```
>>> from django.core.exceptions import ObjectDoesNotExist
>>> try:
...     e = Entry.objects.get(id=3)
...     b = Blog.objects.get(id=1)
... except ObjectDoesNotExist:
...     print "Either the entry or blog doesn't exist."
```

create(**kwargs)

Este método sirve para crear un objeto y guardarlo en un mismo paso. Te permite abreviar dos pasos comunes:

```
>>> p = Person(first_name="Bruce", last_name="Springsteen")
>>> p.save()
```

en una sola línea:

```
>>> p = Person.objects.create(first_name="Bruce", last_name="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 = Person.objects.get(first_name='John', last_name='Lennon')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday=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 = Person.objects.get_or_create(
    first_name = 'John',
    last_name  = 'Lennon',
    defaults   = {'birthday': 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'}
)
```

Nota

Como ya se mencionó, `get_or_create` es utilizado más que nada en scripts que necesiten procesar datos y crear nuevos campos si los que existen no están disponibles. Si necesitas utilizar `get_or_create()` en una vista, por favor asegurate de utilizarlo solo en pedidos `POST` salvo que tengas una buena razón para no hacerlo. Los pedidos `GET` no deberían afectar los datos de ninguna manera; se debe utilizar `POST` en cualquier pedido a una página que pueda tener como efecto secundario una modificación a tus datos.

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.
>>> Entry.objects.count()
4

# Returns the number of entries whose headline contains 'Lennon'
>>> Entry.objects.filter(headline__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_name` como fecha. Este ejemplo devuelve el `Entry` más reciente en la tabla, de acuerdo con el campo `pub_date`:

```
>>> Entry.objects.latest('pub_date')
```

Si el `Meta` de tu modelo especifica `get_latest_by`, se puede omitir el argumento `field_name`. 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.

C.6. 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:

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

se traduce (aproximadamente) al siguiente comando SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '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.

C.6.1. exact

Realiza una búsqueda por coincidencias exactas:

```
>>> Entry.objects.get(headline__exact="Man bites dog")
```

Esto busca objetos que tengan en el campo headline 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) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
```

Esto es por conveniencia, dado que las búsquedas con tipo de búsqueda `exact` son las más frecuentes.

C.6.2. iexact

Realiza una búsqueda por coincidencias exactas sin distinguir mayúsculas de minúsculas:

```
>>> Blog.objects.get(name__iexact='beatles blog')
```

Traerá objetos con nombre `'Beatles Blog'`, `'beatles blog'`, `'BeAtLes BLoG'`, etcétera.

C.6.3. contains

Realiza una búsqueda de subcadenas, distinguiendo mayúsculas y minúsculas:

```
Entry.objects.get(headline__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, por que la abstracción funciona bien. Por ejemplo, para obtener todos los Entries que contengan un símbolo de porciento, simplemente hace falta utilizar el símbolo de porcentaje como cualquier otro caracter:
>
> ```
> Entry.objects.filter(headline__contains='%')
> ```
>
> Django se hace cargo del escapado. El SQL resultante será algo similar a esto:
>
> ```
> SELECT ... WHERE headline LIKE '%\%%';
> ```
>
> Lo mismo vale para el guión bajo. Tanto el símbolo de porcentaje como el guión bajo se deberían manejar de manera transparente.

C.6.4. icontains

Realiza una búsqueda de subcadenas, sin distinguir mayúsculas y minúsculas:

```
>>> Entry.objects.get(headline__icontains='Lennon')
```

A diferencia de `contains`, `icontains` *sí* trerá `today lennon honored`.

C.6.5. gt, gte, lt, and lte

Estos representan los operadores de mayor a, mayor o igual a, menor a, y menor o igual a, respectivamente:

```
>>> Entry.objects.filter(id__gt=4)
>>> Entry.objects.filter(id__lt=15)
>>> Entry.objects.filter(id__gte=0)
```

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").

C.6.6. in

Aplica un filtro para encontrar valores en una lista dada:

```
Entry.objects.filter(id__in=[1, 3, 4])
```

Esto devolverá todos los objetos que tengan un ID de 1, 3 o 4.

C.6.7. startswith

Busca coincidencias de prefijos distinguiendo mayúsculas y minúsculas:

```
>>> Entry.objects.filter(headline__startswith='Will')
```

Esto encontrará los titulares "Will he run¿' y "Willbur named judge", pero no "Who is Will¿' o "will found in crypt".

C.6.8. istartswith

Realiza una búsqueda por prefijos, sin distinguir mayúsculas y minúsculas:

```
>>> Entry.objects.filter(headline__istartswith='will')
```

Esto devolverá los titulares "Will he run¿', "Willbur named judge", y "will found in crypt", pero no "Who is Will¿'

C.6.9. endswith and iendswith

Realiza búsqueda de sufijos, distinguiendo y sin distinguir mayúsculas de minúsculas, respectivamente:

```
>>> Entry.objects.filter(headline__endswith='cats')
>>> Entry.objects.filter(headline__iendswith='cats')
```

C.6.10. range

Realiza una búsqueda por rango:

```
>>> start_date = datetime.date(2005, 1, 1)
>>> end_date = datetime.date(2005, 3, 31)
>>> Entry.objects.filter(pub_date__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.

C.6.11. year, month, and day

Para campos `date` y `datetime`, realiza búsqueda exacta por año, mes o día:

```
# Búsqueda por año
>>>Entry.objects.filter(pub_date__year=2005)

# Búsqueda por mes -- toma enteros
>>> Entry.objects.filter(pub_date__month=12)

# Búsqueda por día
>>> Entry.objects.filter(pub_date__day=3)

# Combinación: devuelve todas las entradas de Navidad de cualquier año
>>> Entry.objects.filter(pub_date__month=12, pub_date_day=25)
```

C.6.12. isnull

Toma valores `True` o `False`, que corresponderán a consultas SQL de `IS NULL`‘‘y ‘‘`IS NOT NULL`, respectivamente:

```
>>> Entry.objects.filter(pub_date__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.

C.6.13. 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`.

C.6.14. 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 `pk` también funcionan con joins. Por ejemplo, estas tres sentencias son equivalentes:

```
>>> Entry.objects.filter(blog__id__exact=3) # Forma explícita
>>> Entry.objects.filter(blog__id=3) # __exact implícito
>>> Entry.objects.filter(blog__pk=3) # __pk implica __id__exact
```

C.7. 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(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

se traduce aproximadamente al siguiente SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '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(pub_date=date(2005, 5, 2)) | Q(pub_date=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(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))
```

no es válido.

Hay algunos ejemplos disponibles online en http://www.djangoproject.com/documentation/0.96/models/or_lookups/ .

C.8. 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 `Entry`, 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 `Entry` a través del atributo `entry_set`: `b.entry_set.all()`.

Todos los ejemplos en esta sección utilizan los modelos de ejemplo `Blog`, `Author` y `Entry` que se definen al principio de esta sección.

C.8.1. Consultas Que Cruzan Relaciones

Django ofrece un mecanismo poderoso e intuitivo para "seguir" relaciones cuando se realizan búsquedas, haciéndose cargo de los `JOIN`s 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 `Entry` que tengan un `Blog` cuyo `nombre` sea `'Beatles Blog'`:

```
>>> Entry.objects.filter(blog__name__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 `Entry` cuyo `headline` contenga `'Lennon'`:

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

C.8.2. Relaciones de Clave Foránea

Si un modelo contiene un `ForeignKey`, las instancias de ese modelo tendrán acceso al objeto relacionado (foráneo) vía un simple atributo del modelo, por ejemplo:

```
e = Entry.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 = Entry.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 = Entry.objects.get(id=2)
e.blog = None
e.save() # "UPDATE blog_entry 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 = Entry.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 = Entry.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".

C.8.3. 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.entry_set.all() # Encontrar todos los objetos Entry relacionados a b.

# b.entry_set es un Manager que devuelve QuerySets.
b.entry_set.filter(headline__contains='Lennon')
b.entry_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 `Entry` fuera cambiado por `blog = ForeignKey(Blog, related_name='entries')`, el ejemplo anterior pasaría a ser así:

```
b = Blog.objects.get(id=1)
b.entries.all() # Encontrar todos los objetos Entry relacionados a b.

# b.entries es un Manager que devuelve QuerySets.
b.entries.filter(headline__contains='Lennon')
b.entries.count()
```

No se puede acceder al `Manager` de `ForeignKey` inverso desde la clase misma; debe ser accedido desde una instancia:

```
Blog.entry_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 = Entry.objects.get(id=234)
  b.entry_set.add(e) # Associates Entry 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.entry_set.create(headline='Hello', body_text='Hi', pub_date=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 = Entry(blog=b, headline='Hello', body_text='Hi', pub_date=datetime.date(2005, 1, 1))
  e.save()
  ```

  Notar 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 `blog` del nuevo `Entry` debería ser `b`.

- `remove(obj1, obj2, ...)`: Quita los objetos indicados del conjunto de objetos relacionados:

  ```
  b = Blog.objects.get(id=1)
  e = Entry.objects.get(id=234)
  b.entry_set.remove(e) # Desasociar al Entry 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, el quitar a `e` de `b.entry_set()` es equivalente a hacer `e.blog = None`, y dado que la definición del campo `ForeignKey blog` (en el modelo `Entry`) 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.entry_set.clear()
  ```

  Notar 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.entry_set = [e1, e2]
```

Si el método `clear()` está definido, todos los objetos pre-existentes serán quitados del `entry_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.

C.8.4. 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 esto lo hará más fácil de entender:

```
e = Entry.objects.get(id=3)
e.authors.all() # Devuelve todos los objetos Author para este Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Devuelve todos los obejtos Entry para este Author.
```

Al igual que los campos `ForeignKey`, los `ManyToManyField` pueden indicar un `related_name`. En el ejemplo anterior, si el campo `ManyToManyField` en el modelo `Entry` indicara `related_name='entries'`, cualquier instancia de `Author` tendría un atributo `entries` en vez de `entry_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.

C.8.5. 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:

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

C.9. Borrando Objectos

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 `Entry` que tengan un año de `pub_date` igual a 2005:

```
Entry.objects.filter(pub_date__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 Entry.
b.delete()
```

Notar 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 `Entry.objects.delete()` y borres *todos* los Entry. Si *realmente* quieres borrar todos los objetos, hay que pedirlo explícitamente al conjunto completo de objetos:

```
Entry.objects.all().delete()
```

C.10. Métodos de Instancia Adicionales

Además de `save()` y `delete()`, un objeto modelo puede tener cualquiera o todos de los siguientes métodos.

C.10.1. 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 "humanamente legible" del campo. Por ejemplo, en el siguiente modelo:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)
class Person(models.Model):
    name = models.CharField(max_length=20)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

cada instancia de `Person` tendrá un método `get_gender_display`:

```
>>> p = Person(name='John', gender='M')
>>> p.save()
>>> p.gender
'M'
>>> p.get_gender_display()
'Male'
```

C.10.2. 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/0.96/models/lookup/.

C.10.3. 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`.

Notar que el campo `ImageField` es técnicamente una subclase de `FileField`, así que todo modelo que tenga un campo `ImageField` obtendrá también este método.

C.10.4. 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.

C.10.5. 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`.)

C.10.6. 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.

C.10.7. 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.

C.11. 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 par simplificar el proceso de escribir vistas. Estas funciones se pueden hallar en el módulo `django.shortcuts`.

C.11.1. 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 funcion 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:

```
# Get the Entry with a primary key of 3
e = get_object_or_404(Entry, 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:

```
# Get the author of blog instance e with a name of 'Fred'
a = get_object_or_404(e.authors, name='Fred')

# Use a custom manager 'recent_entries' in the search for an
# entry with a primary key of 3
e = get_object_or_404(Entry.recent_entries, pk=3)
```

C.11.2. 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.

C.12. 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. Por más instrucciones, véase el Apéndice B..

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 D

Referencia de las vistas genéricas

El Capítulo 9 es una introducción a las vistas genéricas, pero pasa por alto algunos detalles . 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 9 . Tampoco viene mal un repaso a los modelos `Book`, `Publisher` y `Author` definidos en dicho capítulo, ya que serán usados en los ejemplo incluidos en esta apéndice.

D.1. Argumentos comunes a todas las vistas genéricas

La mayoría de las vistas aceptan varios argumentos que pueden modificar su comportamiento. Muchos de esos argumentos funcionan igual para la mayoría de las vistas. La tabla D-1 describe 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.

Cuadro D.1: Argumentos comunes de las vistas genéricas.

Argumento	Descripción
`allow_empty`	Un valor booleano que indica como 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_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. En el **'Capítulo 10'_** se explica con detalle la lista de procesadores de contexto adicionales.
`extra_context`	Un diccionario cuyos valores se añaden al contexto de la plantilla. Si se almacena un objeto que sea invocable, la vista genérica lo ejecutará justo antes de representar la plantilla
`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, `Author.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.
`template_loader`	El cargador de plantillas a utilizar. Por defecto es `django.template.loader`. Véase el **'Capítulo 10'_** donde se da más información acerca de los cargadores de plantillas.
`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`.

Cuadro D.1: Argumentos comunes de las vistas genéricas.

Argumento	Descripción
template_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.

D.2. Vistas genéricas simples

Dentro del módulo `django.views.generic.simple` hay varias vistas sencillas que manejan unos cuantos problemas frecuentes: mostrar una plantilla que no necesita una vista lógica, y hacer una redirección de una página.

D.2.1. Representar una plantilla

Vista a importar: `django.views.generic.simple.direct_to_template`

Esta vista representa una plantilla, a la que se le pasa una variable de plantilla accesible como `{{ params }}`, y que es un diccionario que contiene los parámetros capturados de la URL, si los hubiera.

Ejemplo

Dada la siguiente configuración del URLconf:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    (r'^foo/$',             direct_to_template, {'template': 'foo_index.html'}),
    (r'^foo/(?P<id>\d+)/$', direct_to_template, {'template': 'foo_detail.html'}),
)
```

Una petición a `/foo/` mostraría la plantilla `foo_index.html`, y una solicitud a `/foo/15/` mostraría `foo_detail.html` con una variable de contexto `{{ params.id }}` cuyo valor sería `15`.

Argumentos obligatorios

- `template`: El nombre completo de la plantilla a representar.

D.2.2. Redirigir a otra URL

Vista a importar: `django.views.generic.simple.redirect_to`

Esta vista redirige a otra URL. La URL que se pasa como parámetro puede tener secuencias de formato aptas para diccionarios, que serán interpretadas contra los parámetros capturados desde la URL origen.

Si la URL pasada como parámetro es `None`, Django retornará un mensaje de error 410 ("Gone" según el estándar HTTP).

Ejemplo

Este URLconf redirige desde `/foo/<id>/` a `/bar/<id>/`:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import redirect_to

urlpatterns = patterns('django.views.generic.simple',
    ('^foo/(?p<id>\d+)/$', redirect_to, {'url': '/bar/%(id)s/'}),
)
```

Este ejemplo devuelve una respuesta "Gone" para cualquier petición a `/bar/`:

```
from django.views.generic.simple import redirect_to

urlpatterns = patterns('django.views.generic.simple',
    ('^bar/$', redirect_to, {'url': None}),
)
```

Argumentos obligatorios

- `url`: La URL a la que redirigir, en forma de cadena de texto, o `None` si quereremos devolver una respuesta 410 ("Gone" según el estándar HTTP).

D.3. Vistas de listado/detalle

Las vistas genéricas de listados/detalle (que residen en el módulo `Django.views.generic.list_detail`) 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).

D.3.1. Listas de objetos

Vista a importar: `django.views.generic.list_detail.object_list`
Esta vista sirve para representear una lista de objetos.

Ejemplo

Si consideramos el objeto `Author` tal y como se definió en el capítulo 5, podemos usar la vista `object_list` para obtener un listado sencillo de todos los autores usando el siguiente URLconf:

```
from mysite.books.models import Author
from django.conf.urls.defaults import *
from django.views.generic import list_detail

author_list_info = {
    'queryset' :   Author.objects.all(),
    'allow_empty': True,
}

urlpatterns = patterns('',
    (r'authors/$', list_detail.object_list, author_list_info)
)
```

Argumentos obligatorios

- `queryset`: Un `QuerySet` de los objetos a listar (Véase la table D-1).

Argumentos opcionales

- `paginate_by`: 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.

Además, esta vidta acepta cualquiera de los siguientes argumentos opcionales descritos en la tabla D-1:

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`

- template_loader
- template_name
- template_object_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>_l
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, tendriamos que el queryset sería Author.objects.all(), por lo que la etiqueta de la aplicación será books y el nombre del modelo es author. Con esos datos, el nombre de la plantilla a utilizar por defecto será books/author_list.html.

Contexto de plantilla

Además de los valores que se puedan haber definido en extra_context, el contexto de 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 numero 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:

- Usar un parámetro page en el URLconf. Por ejemplo, tu URLconf podría parecerse a este:

```
(r'^objects/page(?P<page>[0-9]+)/$', 'object_list', dict(info_dict))
```

- Pasar el número de la página mediante un parámetro page en la URL: Por ejemplo, tus URL se podrían parecer a esto:

```
/objects/?page=3
```

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.

D.3.2. Vista de detalle

Vista a importar: django.views.generic.list_detail.object_detail
Esta vista proporciona una representación indidual de los "detalles" de un objeto.

Ejemplo

Siguiendo con el ejemplo anterior, podemos añadir una vista de detalle de cada autor modificacando el URLconf de la siguiente manera:

```
from mysite.books.models import Author
from django.conf.urls.defaults import *
from django.views.generic import list_detail

author_list_info = {
    'queryset' :   Author.objects.all(),
    'allow_empty': True,
}
author_detail_info = {
    "queryset" : Author.objects.all(),
    "template_object_name" : "author",
}

urlpatterns = patterns('',
    (r'authors/$', list_detail.object_list, author_list_info),
    (r'^authors/(?P<object_id>d+)/$', list_detail.object_detail, author_detail_info),
)
```

Argumentos obligatorios

- `queryset`: Un `QuerySet` que será usado para localizar el objeto a mostrar (véase la Tabla D-1).

y luego hace falta, o un:

- `object_id`: El valor de la clave primaria del objeto a mostrar.

o bien:

- `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.

Esta vista también acepta estos argumentos comunes (Véase la tabla D-1):

- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Nombre de la plantilla

Si no se especifican `template_name` ni `template_name_field`, se usará la plantilla `<app_label>/<model_name>_detail.h`

Contexto de plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de la plantilla tendrá los siguientes valores:

- `object`: El objeto. El nombre de esta variable puede ser distinto si se ha especificado el argumento `template_object_name`, cuyo valor es `'object'` por defecto. Si definimos `template_object_name` como `'foo'`, el nombre de la variable será `foo`.

D.4. 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*.

> **Truco:**
> En principio, estas vistas ignoran las fechas que estén situadas en el futuro.
> Esto significa que si intentas visitar una página del 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").

D.4.1. Índice de archivo

Vista a importar: `django.views.generic.date_based.archive_index`

Esta vista proporciona un índice donde se mostraría los "últimos" objetos (es decir, los más recientes) según la fecha.

Ejemplo

Supongamos el típico editor que desea una página con la lista de sus últimos libros publicados. Suponiendo que tenemos un objeto `Book` con un atributo de fecha de publicación, `publication_date`, podemos usar la vista `archive_index` para resolver este problema:

```
from mysite.books.models import Book
from django.conf.urls.defaults import *
from django.views.generic import date_based

book_info = {
    "queryset"   : Book.objects.all(),
    "date_field" : "publication_date"
}

urlpatterns = patterns('',
    (r'^books/$', date_based.archive_index, book_info),
)
```

Argumentos obligatorios

- `date_field`: El nombre de un campo `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.

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.
- `num_latest`: El número de objetos que se deben enviar a la plantilla. Su valor por defecto es 15.

Esta vista también acepta estos argumentos comunes (Véase la tabla D-1):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`

Nombre de la plantilla

Si no se ha especificado `template_name`, se usará la plantilla `<app_label>/<model_name>_archive.html`.

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, 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 mas 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. Por ejemplo, si `num_latest` vale 10, entonces `latest` será una lista de los últimos 10 objetos contenidos en el `queryset`.

D.4.2. Archivos anuales

Vista a importar: `django.views.generic.date_based.archive_year`

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.

Ejemplo

Vamos a ampliar el ejemplo anterior incluyendo una vista que muestre todos los libros publicados en un determinado año:

```
from mysite.books.models import Book
from django.conf.urls.defaults import *
from django.views.generic import date_based

book_info = {
    "queryset"   : Book.objects.all(),
    "date_field" : "publication_date"
}

urlpatterns = patterns('',
    (r'^books/$', date_based.archive_index, book_info),
    (r'^books/(?P<year>d{4})/?$', date_based.archive_year, book_info),
)
```

Argumentos obligatorios

- `date_field`: Igual que en `archive_index` (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 "Contexto 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".

Esta vista también acepta los siguientes argumentos comunes (Véase la Tabla D-1):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Nombre de la plantilla

Si no se especifica ningún valor en `template_name`, la vista usará la plantilla `<app_label>/<model_name>_archive_year.html`.

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, 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.

D.4.3. Archivos mensuales

Vista a importar: `django.views.generic.date_based.archive_month`

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.

Ejemplo

Siguiendo con nuestro ejemplo, añadir una vista mensual debería ser algo sencillo:

```
urlpatterns = patterns('',
    (r'^books/$', date_based.archive_index, book_info),
    (r'^books/(?P<year>d{4})/?$', date_based.archive_year, book_info),
    (
        r'^(?P<year>d{4})/(?P<month>[a-z]{3})/$',
        date_based.archive_month,
        book_info
    ),
)
```

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.

Esta vista también acepta los siguientes argumentos comunes (Véase la Tabla D-1):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Nombre de la plantilla

Si no se especifica ningún valor en `template_name`, la vista usará como plantilla `<app_label>/<model_name>_archive_mon`

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de la plantilla tendrá 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`.

D.4.4. Archivos semanales

Vista a importar: `django.views.generic.date_based.archive_week`
Esta vista muestra todos los objetos de una semana determinada.

> **Nota**
>
> Por consistencia con las bibliotecas de manejo de fechas de Python, Django asume que el primer día de la semana es el domingo.

Ejemplo

```
urlpatterns = patterns('',
    # ...
    (
        r'^(?P<year>d{4})/(?P<week>d{2})/$',
        date_based.archive_week,
        book_info
    ),
)
```

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".

Esta vista también acepta los siguientes argumentos comunes (Véase la Tabla D-1):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

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`

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de la plantilla tendrá 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`.

D.4.5. Archivos diarios

Vista a importar: `django.views.generic.date_based.archive_day`
esta vista muestra todos los objetos para un día determinado.

Ejemplo

```
urlpatterns = patterns('',
    # ...
    (
        r'^(?P<year>d{4})/(?P<month>[a-z]{3})/(?P<day>d{2})/$',
        date_based.archive_day,
        book_info
    ),
)
```

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 seccion 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".

Esta vista también acepta los siguientes argumentos comunes (Véase la Tabla D-1):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Nombre de la plantilla

Si no se ha especificado ningún valor en `template_name` la vista usará como plantilla `<app_label>/<model_name>_archive`

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de 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.

D.4.6. Archivo para hoy

La vista django.views.generic.date_based.archive_today muestra todos los objetos cuya fecha de referencia sea *hoy*. Es exactamente igual a archive_day, excepto que no se utilizan los argumentos year, month ni day, ya que esos datos se obtendrán de la fecha actual.

Ejemplo

```
urlpatterns = patterns('',
    # ...
    (r'^books/today/$', date_based.archive_today, book_info),
)
```

D.4.7. Páginas de detalle basadas en fecha

Vista a importar: django.views.generic.date_based.object_detail

Se usa esta vista para representar un objeto individual.

Esta vista tiene una URL distinta de la vista object_detail; mientras que la última usa una URL como, por ejemplo, /entries/<slug>/, esta usa una URL en la forma /entries/2006/aug/27/<slug>/.

> **Nota**
>
> Si estás usando páginas de detalle basadas en la fecha 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. Lee el apéndice F para más detalles sobre la opción unique_for_date.

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 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.

```
urlpatterns = patterns('',
    # ...
    (
        r'^(?P<year>d{4})/(?P<month>[a-z]{3})/(?P<day>d{2})/(?P<object_id>[w-]+)/$',
        date_based.object_detail,
        book_info
    ),
)
```

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 seccion 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.

  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'`.

Esta vista también acepta los siguientes argumentos comunes (Véase la Tabla D-1):

- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Nombre de la plantilla

Si no se ha especificado ningún valor en `template_name` la vista usará como plantilla `<app_label>/<model_name>_detail.`

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de la plantilla tendrá los siguientes valores:

- `object`: El object. 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`.

D.5. Vistas genericas para Crear/Modificar/Borrar

El módulo `Django.views.generic.create_update` contiene una serie de funciones para crear, modificar y borrar objetos.

> **Nota**
>
> Estas vistas pueden cambiar ligeramente en la nueva revisión de la arquitectura de formularios de Django (que está en fase de desarrollo con el nombre `django.newforms`).

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 9.

D.5.1. Vista de creación de objetos

Vista a importar: `django.views.generic.create_update.create_object`

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, salva el objeto en la base de datos.

Ejemplo

Si quisiéramos permitir al usuario que creara nuevos libros en la base de datos, podríamos hacer algo como esto:

```
from mysite.books.models import Book
from django.conf.urls.defaults import *
from django.views.generic import date_based

book_info = {'model' : Book}

urlpatterns = patterns('',
    (r'^books/create/$', create_update.create_object, book_info),
)
```

Argumentos obligatorios

- `model`: El modelo Django del objeto a crear.

> **Nota**
>
> Obsérvese que esta vista espera el *modelo* del objeto a crear, y no un `QuerySet` como el resto de las vistas anteriores que se han visto previamente.

Argumentos opcionales

- `post_save_redirect`: Una URL, a la cual la vista redirigirá después de haber salvado el objeto. Si no se especifica, se tomará de `object.get_absolute_url()`

  `post_save_redirect`: puede contener cadenas de formato para diccionarios, cuyos valores se interpolarán usando los nombres de los atributos del objeto. Por ejemplo, se podría usar: `post_save_redirect="/polls/%(slug)s`
- `login_required`: Un valor booleano que obliga a que la operación la realice un usuario identificado, ya sea para ver el formulario o para realizar la operación de creación del objeto. Utiliza el subsistema de autentificación y seguridad de Django. Por defecto, su valor es `False`.

  En caso de que se defina como `True`, si un usuario no identificado intentase acceder a esta página o salvar los datos, Django le redirigiría a `/accounts/login/`

Esta vista también acepta los siguientes argumentos comunes (Véase la Tabla D-1):

- `context_processors`
- `extra_context`
- `template_loader`
- `template_name`

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`.

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de la plantilla tendrá los siguientes valores:

- `form`: Una instancia de la clase `FormWrapper`, 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, `name` y `address`:

```
<form action="" method="post">
  <p><label for="id_name">Name:</label> {{ form.name }}</p>
  <p><label for="id_address">Address:</label> {{ form.address }}</p>
</form>
```

Hay que hacer notar que `form` es un FormWrapper definido en *oldforms*, y que no está contemplado en este libro. Véase http://www.djangoproject.com/documentation/0.96/forms/ par más información.

D.5.2. Vista de modificación de objetos

Vista a importar: `django.views.generic.create_update.update_object`

Esta vista es prácticamente igual al anterior, siendo la única diferencia que esta permite la modificación de los atributos del objeto, en vez de su creación.

Ejemplo

Siguiendo con el ejemplo, podemos proporcionar al usuario una interfaz de modificación de los datos de un libro con el siguiente código en el URLconf:

```
from mysite.books.models import Book
from django.conf.urls.defaults import *
from django.views.generic. import date_based

book_info = {'model' : Book}

urlpatterns = patterns('',
    (r'^books/create/$', create_update.create_object, book_info),
    (
        r'^books/edit/(?P<object_id>d+)/$',
        create_update.update_object,
        book_info
    ),
)
```

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`.

Y, o bien un:

- `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 calve 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 D-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.htm`

Contexto de la plantilla

Además de los valores que se puedan haber definido en `extra_context`, el contexto de la plantilla tendrá los siguientes valores:

- `form`: Una instancia de `FormWrapper` que representa el formulario de edición del objeto. Véase la sección "Vista de creación de objetos" para obtener más información de esta variable.
- `object`: El objeto a editar (El nombre de esta variable puede ser diferente si se ha especificado el argumento `template_object_name`).

D.5.3. Vista de borrado de objetos

Vista a importar: `django.views.generic.create_update.delete_object`

Esta vista es muy similar a la dos anteriores: crear y modificar objetos. El propósito de esta 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`.

Apéndice E

Variables de configuración

Tu archivo de configuración contiene toda la configuración de tu instalación de Django. Este apéndice explica cómo funcionan las variables de configuración y qué variables de configuración están disponibles.

> **Nota**
>
> A medida que Django crece, es ocasionalmente necesario agregar o (raramente) cambiar variables de configuración. Debes siempre buscar la información mas reciente en la documentación de configuración en línea que se encuentra en http://www.djangoproject.com/documentation/0.96/settings/.

E.1. 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'
TEMPLATE_DIRS = ('/home/templates/mike', '/home/templates/john')
```

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:

  ```
  MY_SETTING = [str(i) for i in range(30)]
  ```

- El mismo puede importar valores desde otros archivos de configuración.

E.1.1. Valores por omisión

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:

- Carga las variables de configuración desde `global_settings`.
- 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.

E.1.2. Viendo 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 descripto con mas detalle en el Apéndice G.

E.1.3. 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:
    # Do something
```

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 # This won't work.

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.

E.1.4. 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   # Don't do this!
```

El único lugar en el que debes asignar valores a `settings` es en un archivo de configuración.

E.1.5. 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 tu y el usuario de tu servidor Web puedan leerlo. Esto es especialmente importante en un entorno de alojamiento compartido.

E.1.6. Creando tus propias variables de configuración

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.

E.2. 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`).

Consejo:
Puedes encontrar una buena guía acerca de *PYTHONPATH*' en http://diveintopython.org/getting_to_know_python/everything_is_an_object.html.

E.2.1. La utilidad django-admin.py

Cuando usas `django-admin.py` (ver Apéndice G), 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 `startproject` como parte del esqueleto del proyecto asigna un valor a `DJANGO_SETTINGS` en forma automática; revisa el Apéndice G si deseas conocer más sobre `manage.py`.

E.2.2. En el servidor (mod_python)

En tu entorno del servidor activo, necesitarás indicarle a Apache/mod_python qué archivo de configuración debe usar. Haz eso con `SetEnv`:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>
```

Para más información, revisa la documentación en línea de mod_python en http://www.djangoproject.com/documentation/0.96/modpython/.

E.3. Usando variables de configuración sin fijar DJANGO_SETTINGS_MOD

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 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` mas adelante en este apéndice para conocer porqué habría de ocurrir esto). Asumimos que en esos casos ya tienes completo control de tu entorno.

E.3.1. Variables de configuración por omisión personalizados

Si te gustaría que los valores por omisión provinieran desde otra ubicación diferente a `django.conf.global_settings`, puedes pasarle un módulo o 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 sobreescribir 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.

E.3.2. Es necesario que uses 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()` mas 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.

E.4. 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.

E.4.1. ABSOLUTE_URL_OVERRIDES

Valor por omisión: `{}` (diccionario vacío)

Un diccionario interrelacionando cadenas `app_label.model_name` a funciones que toman un objeto modelo y retornan su URL. Esta es una forma de sobreescribir 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),
}
```

Notar 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.

E.4.2. ADMIN_FOR

Valor por omisión: `()` (lista vacía)

Esta variable de configuración es usada en módulos de configuración de sitios de administración. Debe ser una tupla de módulos de configuración (en el formato `'foo.bar.baz'`) para los cuales este sitio es una aplicación de administración.

El sitio de administración usa esto en la documentación automáticamente introspeccionada de modelos, vistas y etiquetas de plantillas.

E.4.3. ADMIN_MEDIA_PREFIX

Valor por omisión: `'/media/'`

Esta variable de configuración es el prefijo de la URL para los medios del sitio de administración: CSS, JavaScript e imágenes. Asegúrate de usar una barra al final.

E.4.4. 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'))
```

Notar que Django el enviará e-mail a *todas* estas personas cada vez que ocurra un error.

E.4.5. 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.

E.4.6. 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 `CommonMiddleware` (ver Capítulo 15). Ver también `PREPEND_WWW`.

E.4.7. CACHE_BACKEND

Valor por omisión: `'simple://'`

El ***back-end*** de cache a usarse (ver Capítulo 13).

E.4.8. 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 13).

E.4.9. DATABASE_ENGINE

Valor por omisión: `''` (cadena vacía)

Esta variable de configuración indica qué ***back-end*** de base de datos debe usarse: `'postgresql_psycopg2'`, `'postgresql'`, `'mysql'`, `'mysql_old'` o `'sqlite3'`.

E.4.10. DATABASE_HOST

Valor por omisión: `''` (cadena vacía)

Esta variable de configuración indica qué equipo 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:

```
DATABASE_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.

E.4.11. DATABASE_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.

E.4.12. DATABASE_OPTIONS

Valor por omisión: `{}` (diccionario vacío)
Parámetros extra a usarse cuando se establece la conexión a la base de datos. Consulta el módulo back-end para conocer las palabras claves disponibles.

E.4.13. DATABASE_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.

E.4.14. DATABASE_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.

E.4.15. DATABASE_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 da datos. No se usa con SQLite.

E.4.16. 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 Apéndice F, Tabla F-2).
Ver también `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` y `MONTH_DAY_FORMAT`.

E.4.17. 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`.

E.4.18. 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 ***backtraces*** sin ver variables de configuración con contenido importante (u ofensivo).
Si 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.

E.4.19. 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`. Ver el Apéndice H para más información acerca de los objetos `HttpResponse`.

E.4.20. 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`.

E.4.21. 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.

E.4.22. 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 15).

E.4.23. EMAIL_HOST

Valor por omisión: `'localhost'`

El host a usarse para enviar e-mail. Ver también `EMAIL_PORT`.

E.4.24. 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`.

E.4.25. 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`.

E.4.26. EMAIL_PORT

Valor por omisión: `25`

El puerto a usarse pata el servidor SMTP definido en `EMAIL_HOST`.

E.4.27. 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_m`
Probablemente querrás incluir un espacio al final.

E.4.28. 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 se buscará en las mismas. Notar que esas rutas deben usar barras de estilo Unix aún en Windows. Es usado por el framework de pruebas de Django, el cual se trata en http://www.djangoproject.com/documentation/0.96/testing/.

E.4.29. IGNORABLE_404_ENDS

Valor por omisión: `('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi', 'favicon.ico', '.php')`

Ver también `IGNORABLE_404_STARTS` y `Error reporting via e-mail`.

E.4.30. IGNORABLE_404_STARTS

Valor por omisión: `('/cgi-bin/', '/_vti_bin', '/_vti_inf')`

Una tupla de cadenas que especifican las partes iniciales de URLs que deben ser ignoradas para el envío de mensajes de e-mail para errores 404. Ver también `SEND_BROKEN_LINK_EMAILS` y `IGNORABLE_404_ENDS`.

E.4.31. 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 un paquete Python que contiene una aplicación Django. Ver el Capítulo 5 para más información acerca de aplicaciones.

E.4.32. 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` (ver Capítulo 15)

E.4.33. JING_PATH

Valor por omisión: `'/usr/bin/jing'`

La ruta al ejecutable Jing. Jing es un validador RELAX NG, y Django usa el mismo para validar los campos `XMLField` en tus modelos. Ver http://www.thaiopensource.com/relaxng/jing.html.

E.4.34. LANGUAGE_CODE

Valor por omisión: `'en-us'`

Una cadena representando el código de idioma para esta instalación. Debe estar expresado usando el formato estándar -- por ejemplo, Inglés de EUA es "en-us". Ver el Capítulo 18.

E.4.35. 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`.

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 18 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.

E.4.36. 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`.

E.4.37. MEDIA_ROOT

Valor por omisión: `''` (cadena vacía)

Una ruta absoluta al directorio que contiene medios para esta instalación (por ej. `"/home/media/media.lawrence.com/"`). Ver también `MEDIA_URL`.

E.4.38. MEDIA_URL

Valor por omisión: `''` (cadena vacía)

Esta URL maneja los medios servidos desde `MEDIA_ROOT` (por ej. `"http://media.lawrence.com"`).

Notar 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"`

E.4.39. MIDDLEWARE_CLASSES

Valor por omisión:

```
("django.contrib.sessions.middleware.SessionMiddleware",
 "django.contrib.auth.middleware.AuthenticationMiddleware",
 "django.middleware.common.CommonMiddleware",
 "django.middleware.doc.XViewMiddleware")
```

Una tupla de clases middleware a usarse. Ver el Capítulo 15.

E.4.40. 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` (ver Apéndice F, tabla F-2)

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`.

E.4.41. 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 15). Ver también `APPEND_SLASH`.

E.4.42. PROFANITIES_LIST

Una tupla de profanidades, como cadenas, que dispararán un error de validación cuando se llama al validador `hasNoProfanities`.

No enumeramos aquí los valores por omisión, debido a que esto podría hacer que nos cayera encima la comisión de clasificación de la MPAA. Para ver los valores por omisión examina el archivo `django/conf/global_settings.py`.

E.4.43. 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.

E.4.44. 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 mas larga mejor. `django-admin startproject` crea una en forma automática y en la mayoría de los casos no será necesario que la modifiques.

E.4.45. 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 referer no vacía (en otras palabras un enlace roto). Es solamente usado si está instalado `CommonMiddleware` (ver Capítulo 15). Ver también `IGNORABLE_404_STARTS` y `IGNORABLE_404_ENDS`.

E.4.46. SERIALIZATION_MODULES

Valor por omisión: No definida.

La serialización es una característica que todavía está bajo fuerte desarrollo. Revisa la documentación en línea en http://www.djangoproject.com/documentation/0.96/serialization/ si deseas más información.

E.4.47. 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` y `MANAGERS`.

E.4.48. SESSION_COOKIE_AGE

Valor por omisión: `1209600` (dos semanas, en segundos)

Esta es la edad de las cookies de sesión, en segundos. Ver Capítulo 12.

E.4.49. 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 12.

E.4.50. 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 12.

E.4.51. 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 12.

E.4.52. 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.

E.4.53. SESSION_SAVE_EVERY_REQUEST

Valor por omisión: `False`

Esta variable de configuración indica si la sesión debe ser grabada en cada petición. Ver Capítulo 12.

E.4.54. 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.

E.4.55. TEMPLATE_CONTEXT_PROCESSORS

Valor por omisión:

```
("django.core.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n")
```

Una tupla de ***callables*** que son usados para poblar el contexto en `RequestContext`. Esos ***callables*** reciben como argumento un objeto petición y retornan un diccionario de items a ser fusionados con el contexto. Ver **'Capítulo 10'_**.

E.4.56. 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.

Notar 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`.

E.4.57. 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 Capítulos 4 y 10.

E.4.58. TEMPLATE_LOADERS

Valor por omisión: `('django.template.loaders.filesystem.load_template_source',)`

Una tupla de ***callables*** (como cadenas) que saben como importar plantillas desde diferentes orígenes. Ver **'Capítulo 10'_**.

E.4.59. 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.

E.4.60. 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/documentation/0.96/testing/.

E.4.61. TEST_DATABASE_NAME

Valor por omisión: `None`

El nombre de la base de datos a usarse cuando se ejecute la batería de pruebas (por *test suite*). Si se especifica un valor `None`, el nombre de la base de datos de pruebas será `'test_' + settings.DATABASE_NAME`. Ver la documentación del framework de pruebas de Django el cual se describe en línea en http://www.djangoproject.com/documentation/0.96/testing/.

E.4.62. 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`.

E.4.63. TIME_ZONE

Valor por omisión: `'America/Chicago'`

Una cadena que representa la zona horaria para esta instalación. Las zonas horarias se expresan en el formato `zic` estándar de Unix. Puede encontrarse una lista relativamente completa de cadenas de zonas horarias en http://www.postgresql.org/docs/8.1/static/datetime-keywords.html#DATETIME-TIMEZONE-SET-TABLE.

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.

E.4.64. URL_VALIDATOR_USER_AGENT

Valor por omisión: `Django/<version> (http://www.djangoproject.com/)`

La cadena usarse como la cabecera `User-Agent` cuando se realizan verificaciones acerca e si las URLs existen (ver la opción `verify_exists` de `URLField`; ver Apéndice B).

E.4.65. 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 15).

E.4.66. USE_I18N

Valor por omisión: `True`

Este Booleano especifica si debe activarse el sistema de internacionalización de Django (ver Capítulo 18). 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.

E.4.67. YEAR_MONTH_FORMAT

Valor por omisión: `'F Y'`

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- 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".

Ver también `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT` y `MONTH_DAY_FORMAT`.

Apéndice F

Etiquetas de plantilla y filtros predefinidos

En el Capítulo 4 se hace una introducción de las etiquetas de plantilla y filtros más utilizados, pero Django incorpora muchas más. En este apéndice se listan todas 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.

F.1. Etiquetas predefinidas

F.1.1. block

Define un bloque que puede ser sobreescrito por las plantillas derivadas. Véase la sección acerca de herencia de plantillas en el Capítulo 4 para más información.

F.1.2. comment

Ignora todo lo que aparezca entre `{% comment %}` y `{% endcomment %}`.

F.1.3. cycle

Rota una cadena de texto entre diferentes valores, cada vez que aparece la etiqueta.

Dentro de un bucle, el valor rotan 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.

F.1.4. debug

Muestra un montón de información para depuración de errores, incluyendo el contexto actual y los módulos importados.

F.1.5. 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 el Capítulo 4 podrás encontrar muchos ejemplo de uso de esta etiqueta.

F.1.6. 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 %}
    This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

F.1.7. 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 %}
```

F.1.8. for

Itera sobre cada uno de los elementos de un lista o *array*. Por ejemplo, para mostrar una lista de atletas, cuyos nombres estén en la lista `athlete_list`, podríamos hacer:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</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 F-1.

Cuadro F.1: Variables accesibles dentro de bucles { % for %}

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.

F.1.9. if

La etiqueta `{% if %}` evalua una variable. Si dicha variable se evalua como una expresión "verdadera" (Es decir, que el valor exista, no esté vacia y no es el valor booleano `False`), se muestra el contenido del bloque:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}
```

Si la lista `athlete_list` no está vacía, podemos mostrar el número de atletas con la expresión `{{ athlete_list|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.

Las etiquetas `if` pueden usar operadores lógicos como `and`, `or` y `not` para evaluar expresiones más complejas:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches (OK, so
    writing English translations of Boolean logic sounds
    stupid; it's not our fault).
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% 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 athlete_list and coach_list or cheerleader_list %}
```

Para combinar operadores `and` y `or`, puedes usar sentencias `if` anidadas, como en el siguiente ejemplo:

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% 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 athlete_list or coach_list or parent_list or teacher_list %}
```

F.1.10. ifchanged

Comprueba si un valor ha sido cambiado desde la última iteración de un bucle.
La etiqueta `ifchanged` solo tiene sentido dentro de un bucle. Tiene dos usos posibles:

1. Comprueba su propio contenido mostrado contra su estado anterior, y solo lo muestra si el contenido ha cambiado. El siguiente ejemplo muestra una lista de días, y solo aparecerá el nombre del mes si este cambia:

```
<h1>Archive for {{ 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. Se le pasa una o más variables, y se comprueba si esas variables han sido cambiadas:

```
{% for date in days %}
    {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
    {% ifchanged date.hour date.date %}
        {{ date.hour }}
    {% endifchanged %}
{% endfor %}
```

   El ejemplo anterior muestra la fecha cada vez que cambia, pero sólo muestra la hora si tanto la hora como el día han cambiado.

F.1.11. 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 %}
```

Sólo se puede comprobar la igualdad de variables o cadenas de texto. No se puede comparar con objetos Python como `True` o `False`. Para ello, utilice la etiqueta `if` directamente.

F.1.12. ifnotequal

Es igual que `ifequal`, excepto que comprueba que los dos parámetros suministrados *no* sean iguales.

F.1.13. 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%}
```

F.1.14. load

Carga una biblioteca de plantillas. En el **'Capítulo 10'**_ puedes encontrar más información acerca de las bibliotecas de plantillas.

F.1.15. 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.

La tabla F-2 muestra las cadenas de formato que se pueden utilizar.

Cuadro F.2: Cadenas de formato para fechas y horas

Carác. formato	Descripción	Ejemplo de salida
a	`'a.m.'` o `'p.m.'`. (Obsérvese que la salida es ligeramente distinta de la de PHP, ya que aquí se incluyen puntos para adecuarse al libro de estilo de Associated Press).	`'a.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'`

Cuadro F.2: Cadenas de formato para fechas y horas

Carác. formato	Descripción	Ejemplo de salida
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. Además, se usarán las cadenas de texto especiales `'midnight'` y `'noon'` para la medianoche y el mediodía respectivamente.	`'1 a.m.'`, `'1:30 p.m.'` , `'midnight'`, `'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

He aquí un ejemplo:

```
It is {% 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:

```
It is the {% now "jS o\f F"%}
```

El ejemplo mostraría: "It is the 4th of September".

F.1.16. regroup

Reagrupa una lista de objetos similares usando un atributo común.

Para comprender esta etiqueta, es mejor recurrir a un ejemplo. Digamos que `people` es una lista de objetos de tipo `Person`, y que dichos objetos tienen los atributos `first_name`, `last_name` y `gender`. Queremos mostrar un listado como el siguiente:

```
* Male:
    * George Bush
    * Bill Clinton
* Female:
    * Margaret Thatcher
    * Condoleezza Rice
* Unknown:
    * Pat Smith
```

El siguiente fragmento de plantilla mostraría como realizar esta tarea:

```
{% regroup people by gender 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 people|dictsort:"gender" by gender as grouped%}
```

F.1.17. 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>
        Hello
    </strong>
{% endspaceless %}
```

F.1.18. 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.

F.1.19. 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 F-3.

Cuadro F.3: Argumentos válidos de templatetag

Argumento	Salida
`openblock`	`{%`
`closeblock`	`%}`
`openvariable`	`{{`
`closevariable`	`}}`
`openbrace`	`{`
`closebrace`	`}`
`opencomment`	`{#`
`closecomment`	`#}`

F.1.20. 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 nuestras plantillas:

```
{% url path.to.some_view arg1,arg2,name1=value1 %}
```

El primer argumento es la ruta a la función de vista, en el formato `paquete.paquete.modulo.funcion`. 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.

Por ejemplo, supongamos que tenemos una vista, `app_name.client`, y que en el URLconf se la indica que acepta un parámetro, el identificador del cliente. La línea del URL podría ser algo así:

```
('^client/(\d+)/$', 'app_name.client')
```

Si este URLconf fuera incluido en el URLconf del proyecto bajo un directorio, como en este ejemplo:

```
('^clients/', include('project_name.app_name.urls'))
```

Podríamos crear un enlace a esta vista, en nuestra plantilla, con la siguiente etiqueta:

```
{% url app_name.client client.id %}
```

La salida de esta etiqueta será `/clients/client/123/`.

F.1.21. 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.gif" height="10" width="{% widthratio this_value max_value 100 %}" />
```

Si `this_value` vale 175 y `max_value` 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).

F.2. Filtros predefinidos

F.2.1. add

Ejemplo:

```
{{ value|add:"5" }}
```

Suma el argumento indicado.

F.2.2. addslashes

Ejemplo:

```
{{ string|addslashes }}
```

Añade barras invertidas antes de las comillas, ya sean simples o dobles. Es útil para pasar cadenas de texto como javascript, por ejemplo:

F.2.3. capfirst

Ejemplo:

```
{{ string|capfirst }}
```

Pasa a mayúsculas la primera letra de la primera palabra.

F.2.4. center

Ejemplo:

```
{{ string|center:"50" }}
```

Centra el texto en un campo de la anchura indicada.

F.2.5. cut

Ejemplo:

```
{{ string|cut:"spam" }}
```

Elimina todas las apariciones del valor indicado.

F.2.6. date

Ejemplo:

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

Formatea una fecha de acuerdo al formato indicado en la cadena de texto (Se usa el mismo formato que con la etiqueta `now`).

F.2.7. default

Ejemplo:

```
{{ value|default:"(N/A)" }}
```

Si `value` no está definido, se usa el valor del argumento en su lugar.

F.2.8. default_if_none

Ejemplo:

```
{{ value|default_if_none:"(N/A)" }}
```

Si `value` es nulo, se usa el valor del argumento en su lugar.

F.2.9. dictsort

Ejemplo:

```
{{ list|dictsort:"foo" }}
```

Acepta una lista de diccionarios y devuelve una lista ordenada según la propiedad indicada en el argumento.

F.2.10. dictsortreversed

Ejemplo:

```
{{ list|dictsortreversed:"foo" }}
```

Acepta una lista de diccionarios y devuelve una lista ordenada de forma descendente según la propiedad indicada en el argumento.

F.2.11. divisibleby

Ejemplo:

```
{% if value|divisibleby:"2"%}
    Even!
{% else%}
    Odd!
{% else%}
```

Devuelve `True` si es valor pasado es divisible por el argumento.

F.2.12. escape

Ejemplo:

```
{{ string|escape }}
```

Transforma un texto que esté en HTML de forma que se pueda representar en una página web. Concretamente, realiza los siguientes cambios:

- `"&"` a `"&"`
- `<` a `"<"`
- `>` a `">"`
- `'"'` (comilla doble) a `'"'`
- `"'"` (comillas simple) a `'''`

F.2.13. filesizeformat

Ejemplo:

```
{{ value|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.

F.2.14. first

Ejemplo:

```
{{ list|first }}
```

Devuelve el primer elemento de una lista.

F.2.15. fix_ampersands

Ejemplo:

```
{{ string|fix_ampersands }}
```

Reemplaza los símbolos *ampersand* con la entidad `&`.

F.2.16. floatformat

Ejemplos:

```
{{ value|floatformat }}
{{ value|floatformat:"2" }}
```

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:

- `36.123` se representaría como `36.1`.
- `36.15` se representaría como `36.2`.
- `36` se representaría como `36`.

Si te utiliza un argumento numérico, `floatformat` redondea a ese número de lugares decimales:

- `36.1234` con floatformat:3 se representaría como `36.123`.
- `36` con floatformat:4 se representaría como `36.0000`.

Si el argumento pasado a `floatformat` es negativo, redondeará a ese número de decimales, pero sólo si el número tiene parte decimal.

- `36.1234` con floatformat:-3 gets converted to `36.123`.
- `36` con floatformat:-4 gets converted to `36`.

Usar `floatformat` sin argumentos es equivalente a usarlo con un argumento de -1.

F.2.17. get_digit

Ejemplo:

```
{{ value|get_digit:"1" }}
```

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.

F.2.18. join

Ejemplo:

```
{{ list|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)`

F.2.19. length

Ejemplo:

```
{{ list|length }}
```

Devuelve la longitud del valor.

F.2.20. length_is

Ejemplo:

```
{% if list|length_is:"3"%}
    ...
{% endif %}
```

Devuelve un valor booleano que será verdadero si la longitud de la entrada coincide con el argumento suministrado.

F.2.21. linebreaks

Ejemplo:

```
{{ string|linebreaks }}
```

Convierte los saltos de línea en etiquetas `<p>` y `<br />`.

F.2.22. linebreaksbr

Ejemplo:

```
{{ string|linebreaksbr }}
```

Convierte los saltos de línea en etiquetas `<br />`.

F.2.23. linenumbers

Ejemplo:

```
{{ string|linenumbers }}
```

Muestra el texto de la entrada con números de línea.

F.2.24. ljust

Ejemplo:

```
{{ string|ljust:"50" }}
```

Justifica el texto de la entrada a la izquierda utilizando la anchura indicada.

F.2.25. lower

Ejemplo:

```
{{ string|lower }}
```

Convierte el texto de la entrada a letras minúsculas.

F.2.26. make_list

Ejemplo:

```
{% for i in number|make_list%}
    ...
{% endfor%}
```

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.

F.2.27. phone2numeric

Ejemplo:

```
{{ string|phone2numeric }}
```

Convierte un número de teléfono (que incluso puede contener letras) a su forma numérica equivalente. Por ejemplo `'800-COLLECT'` se transformará en `'800-2655328'`.

La entrada no tiene porque ser un número de teléfono válido. El filtro convertirá alegremente cualquier texto que se le pase.

F.2.28. pluralize

Ejemplo:

```
The list has {{ list|length }} item{{ list|pluralize }}.
```

Retorno el sufijo para formar el plural si el valor es mayor que uno. Por defecto el sufijo es `'s'`.

Ejemplo:

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

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:

```
You have {{ num_walruses }} walrus{{ num_walrus|pluralize:"es" }}.
```

Para aquellas palabras que forman el plural de forma más compleja que con un simple sufijo, hay otra tercera sintaxis que permite indicar las formas en singular y en plural a partir de una raíz común.

Ejemplo:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

F.2.29. pprint

Ejemplo:

```
{{ object|pprint }}
```

Un recubrimiento que permite llamar a la función de Python `pprint.pprint`. Se usa sobre todo para tareas de depurado de errores.

F.2.30. random

Ejemplo:

```
{{ list|random }}
```

Devuelve un elemento elegido al azar de la lista.

F.2.31. removetags

Ejemplo:

```
{{ string|removetags:"br p div" }}
```

Elimina de la entrada una o varias clases de etiquetas [X]HTML. Las etiquetas se indican en forma de texto, separando cada etiqueta a eliminar por un espacio.

F.2.32. rjust

Ejemplo:

```
{{ string|rjust:"50" }}
```

Justifica el texto de la entrada a la derecha utilizando la anchura indicada..

F.2.33. slice

Ejemplo:

```
{{ some_list|slice:":2" }}
```

Devuelve una sección de la lista.

Usa la misma sintaxis que se usa en Python para seccionar una lista. Véase http://diveintopython.org/native_data_types/lists.ht para una explicación.

F.2.34. slugify

Ejemplo:

```
{{ string|slugify }}
```

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.

F.2.35. stringformat

Ejemplo:

```
{{ number|stringformat:"02i" }}
```

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.

En `http://docs.python.org/lib/typesseq-strings.html` puedes consultar las opciones de formateo de cadenas de Python.

F.2.36. striptags

Ejemplo:

```
{{ string|striptags }}
```

Elimina todas las etiquetas [X]HTML.

F.2.37. time

Ejemplo:

```
{{ value|time:"P" }}
```

Formatea la salida asumiendo que es una fecha/hora, con el formato indicado como argumento (Lo mismo que la etiqueta `now`).

F.2.38. timesince

Ejemplos:

```
{{ datetime|timesince }}
{{ datetime|timesince:"other_datetime" }}
```

Representa una fecha como un intervalo de tiempo (por ejemplo, "4 days, 6 hours").

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 `{{ comment_date|timesince:blog_date }}` devolvería "8 hours".

F.2.39. timeuntil

Ejemplos:

```
{{ datetime|timeuntil }}
{{ datetime|timeuntil:"other_datetime" }}
```

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 }}` devolvería "28 days".

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 }}` devolvería "7 days".

F.2.40. title

Ejemplo:

```
{{ string|titlecase }}
```

Representa una cadena de texto en forma de título, siguiendo las convenciones del idioma inglés (todas las palabras con la inicial en mayúscula).

F.2.41. truncatewords

Ejemplo:

```
{{ string|truncatewords:"15" }}
```

Recorta la salida de forma que tenga como máximo el número de palabras que se indican en el argumento.

F.2.42. truncatewords_html

Ejemplo:

```
{{ string|truncatewords_html:"15" }}
```

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 usada solamente si sabemos que en la entrada va texto HTML.

F.2.43. unordered_list

Ejemplo:

```
<ul>
    {{ list|unordered_list }}
</ul>
```

Acepta una lista, e incluso varias listas anidadas, y recorre recursivamente las mismas representándolas en forma de listas HTML no ordenadas, *sin incluir* las etiquetas de inicio y fin de lista (`<ul>` y `</ul>` respectivamente).

Se asume que las listas está en el formato correcto. Por ejemplo, si `var` contiene `['States', [['Kansas', [['Lawrence', []], ['Topeka', []]]], ['Illinois', []]]]`, entonces `{{ var|unordered_list }}` retornaría lo siguiente:

```
<li>States
<ul>
        <li>Kansas
        <ul>
                <li>Lawrence</li>
                <li>Topeka</li>
        </ul>
        </li>
        <li>Illinois</li>
</ul>
</li>
```

F.2.44. upper

Ejemplo:

```
{{ string|upper }}
```

Convierte una string a mayúsculas.

F.2.45. urlencode

Ejemplo:

```
<a href="{{ link|urlencode }}">linkage</a>
```

Escapa la entrada de forma que pueda ser utilizado dentro de una URL.

F.2.46. urlize

Ejemplo:

```
{{ string|urlize }}
```

Transforma un texto de entrada, de forma que si contiene direcciones URL en texto plano, las convierte en enlaces HTML.

F.2.47. urlizetrunc

Ejemplo:

```
{{ string|urlizetrunc:"30" }}
```

Convierte las direcciones URL de un texto en enlaces, recortando la representación de la URL para que el número de caracteres sea como máximo el del argumento suministrado.

F.2.48. wordcount

Ejemplo:

```
{{ string|wordcount }}
```

Devuelve el número de palabras en la entrada.

F.2.49. wordwrap

Ejemplo:

```
{{ string|wordwrap:"75" }}
```

Ajusta la longitud del texto para las líneas se adecúen a la longitud especificada como argumento.

F.2.50. yesno

Ejemplo:

```
{{ boolean|yesno:"Yes,No,Perhaps" }}
```

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 tabla F-4.

Cuadro F.4: Ejemplos del filtro yesno

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`.

Apéndice G

El utilitario django-admin

`django-admin.py` es el utilitario de linea de comandos de Djangopara 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 wrapper liviano en torno a `django-admin.py`. Toma cuidado de dos cosas por ti antes de delegar 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.

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 ~TRA Panel de Control ~TRA Sistema ~TRA Avanzado ~TRA 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.

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`.

G.1. Uso

El uso básico es:

```
django-admin.py action [options]
```

o:

```
manage.py action [options]
```

`action` debe ser una de las acciones listadas en este documento. `options`, que es opcional, deben ser cero o más opciones de las listadas en este documento.

Ejecuta `django-admin.py --help` para ver un mensaje de ayuda que incluye una larga lista de todas las opciones y acciones disponibles.

La mayoría de las acciones toman una lista de nombres de aplicación. Un *nombre de aplicación* es el nombre base del paquete que contiene tus modelos. Por ejemplo, si tu `INSTALLED_APPS` contiene el string `'mysite.blog'`, el nombre de la aplicación es `blog`.

G.2. Acciones Disponibles

Las siguientes secciones cubren las acciones disponibles.

G.2.1. adminindex [appname appname ...]

Imprime el snippet de la plantilla de admin-index para los nombres de aplicación dados. Usa los snippets de la plantilla de admin-index si quiere personalizar la apariencia de la página del índice del administrador.

G.2.2. createcachetable [tablename]

Crea una tabla de cache llamada `tablename` para usar con el back-end de cache de la base de datos. Ver el Capítulo 13 para más acerca de caching.

G.2.3. 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.

G.2.4. 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 `"###"`.

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.

G.2.5. dumpdata [appname appname ...]

Dirige a la salida estándar todos los datos de la base de datos asociados con la(s) aplicación(es) nombrada(s).

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). La opción `--indent` puede ser usada para lograr una impresión diseñada de la salida.

Si no se provee niingún nombre de aplicación, se volcarán todas las aplicaciones instaladas.

La salida de `dumpdata` puede usarse como entrada para `loaddata`.

G.2.6. flush

Devuelve la base de datos al estado en el que estaba inmediatemente después de que se ejecutó syncdb. Esto significa que todos los datos serán eliminados de la base de datos, todo manejador de postsinchronización será reejecutado, y el componente `initial_data` será reinstalado.

G.2.7. inspectdb

Realiza la introspección sobre las tablas de la base de datos apuntada por la configuración `DATABASE_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 genrados para personalizarlos. En particular, necesitarás reordenar los modelos de manera tal que las relaciones esten 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.

G.2.8. loaddata [fixture fixture ...]

Busca y carga el contenido del 'fixture' nombrado en la base de datos.

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:

- En el directorio `fixtures` de cada aplicación instalada.
- En todo directorio nombreado en la configuración `FIXTURE_DIRS`
- En el path literal nombrado por el fixture

Django cargará todos los fixtures que encuentre en estas ubucaciones 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`.

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 tables, 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.

G.2.9. reset [appname appname ...]

Ejecuta el equivalente de `sqlreset` para los nombres de aplicación dados.

G.2.10. runfcgi [options]

Inicia un conjunto de procesos FastCGI adecuados para su uso con cualquier servidor Web que admita el protocolo FastCGI. Ver Capítulo 20 para más información acerca del desarrollo bajo FastCGI.

Este comando requiere el módulo Python FastCGI de `flup` (`http://www.djangoproject.com/r/flup/`).

G.2.11. runserver [número de puerto opcional, or direcciónIP:puerto]

Inicia un servidor Web liviano de desarollo en la máquina local. machine. Por omisión, el servidor 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 relizado 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 7000
```

O para ejecutar el servidos en el puerto 7000 en la dirección IP 1.2.3.4, usa esto:

```
django-admin.py runserver 1.2.3.4:7000
```

Sirviendo Archivos Estáticos con el Servidor de Desarrollo

Por omisión, el servidor de desarrollo no sirve archivos estáticos para tu sitio (como archivos CSS, imágenes, cosas bajo `MEDIA_ROOT_URL`, etc.). Si quieres configurar Django para servir medios estáticos, lee acerca de esto en http://www.djangoproject.com/documentation/0.96/static_files/.

Deshabilitando Autoreload

Para deshabilitar la recarga automática del código mientras el servidor de desarrollo se ejecuta, usa la opción `--noreload`, como en:

```
django-admin.py runserver --noreload
```

G.2.12. shell

Inicia el intérprete interactivo de Python.

Django utilizará IPython (`http://ipython.scipy.org/`) si no está instalado. Si tienes IPython instalado y quieres forzar el uso del intérprete Python "plano", usa la opción `--plain`, como en:

```
django-admin.py shell --plain
```

G.2.13. sql [appname appname ...]

Imprime las sentencias SQL `CREATE TABLE` para las aplicaciones mencionadas.

G.2.14. sqlall [appname appname ...]

Imprime las sentencias SQL `CREATE TABLE` y los datos iniciales para las aplicaciones mencionadas.
Busca en la descripción de `sqlcustom` para una explicación de como especificar los datos iniciales.

G.2.15. sqlclear [appname appname ...]

Imprime las sentencias SQL `DROP TABLE` para las aplicaciones mencionadas.

G.2.16. sqlcustom [appname appname ...]

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.

G.2.17. sqlindexes [appname appname ...]

Imprime las sentencias SQL `CREATE INDEX` para las aplicaciones mencionadas.

G.2.18. sqlreset [appname appname ...]

Imprime las sentencias SQL `DROP TABLE` seguidas de las `CREATE TABLE` para las aplicaciones mencionadas.

G.2.19. sqlsequencereset [appname appname ...]

Imprime las sentencias SQL para reinicializar las secuencias de las aplicaciones mencionadas.

Necesitarás esta SQL solo si estás usando PostgreSQL y has insertado datos a mano. Cuando haces eso, las secuencias de las claves primarias de PostgreSQL pueden quedar fuera de sincronismo con las que están en la base de datos, y las SQL que genera este comando las limpiarán.

G.2.20. startapp [appname]

Crea una estructura de directorios para una aplicación Django con el nombre de aplicación dado, en el directorio actual.

G.2.21. startproject [projectname]

Crea una estructura de directorios Django para el nombre de proyecto dado, en el directorio actual.

G.2.22. syncdb

Crea las tablas de la base de datos para todas las aplicaciones en `INSTALLED_APPS` cuyas tablas aún no hayan sido creadas.

Usa este comando cuando hayas agregado nuevas aplicaciones a tu proyecto y quieras instalarlas en la base de datos. Esto incluye cualquier aplicación incorporada en Django que esté en `INSTALLED_APPS` por omisión. Cuando empieces un nuevo proyecto, ejecuta este comando para instalas las aplicaciones predeterminadas.

Si estás instalando la aplicación `django.contrib.auth`, `syncdb` te dará la opción de crear un superusuario inmediatamente. `syncdb` también buscará e instalará algun fixture llamado `initial_data`. Ver la documentación de `loaddata` para los detalles de la especificación de los archivos de datos de fixture.

G.2.23. test

Descubre y ejecuta las pruebas para todos los modelos instalados. El testeo aún está en desarrollo mientras se escribe este libro, así que para aprender más necesitarás leer la documentación online en http://www.djangoproject.com/documentation/0.96/testing/.

G.2.24. validate

Valida todos los modelos instalados (según la configuración de `INSTALLED_APPS`) e imprime errores de validación en la salida estándar.

G.3. Opciones Disponibles

Las secciones que siguen delinean las opciones que puede tomar `django-admin.py`.

G.3.1. --settings

Ejemplo de uso:

```
django-admin.py syncdb --settings=mysite.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`.

Observa que esta opción no es necesaria en `manage.py`, ya que toma en cuenta la configuración de `DJANGO_SETTINGS_MODULE` por tí.

G.3.2. --pythonpath

Ejemplo de uso:

```
django-admin.py syncdb --pythonpath='/home/djangoprojects/myproject'
```

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`.

Observa que esta opción no es necesaria en `manage.py`, ya que tiene cuidado de configurar la ruta de Python por tí.

G.3.3. --format

Ejemplo de uso:

```
django-admin.py dumpdata --format=xml
```

Especifica el formato de salida que será utilizado. El nombre provisto debe ser el nombre de un serializador registrado.

G.3.4. --help

Muestra un mensaje de ayuda que incluye una larga lista de todas las opciones y acciones disponibles.

G.3.5. --indent

Ejemplo de uso:

```
django-admin.py dumpdata --indent=4
```

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.

G.3.6. --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.

G.3.7. --noreload

Deshabilita el uso del autoreloader cuando se ejecuta el servidor de desarrollo.

G.3.8. --version

Muestra la versión actual de Django.
Ejemplo de salida:

```
0.9.1
0.9.1 (SVN)
```

G.3.9. --verbosity

Ejemplo de uso:

```
django-admin.py syncdb --verbosity=2
```

Determina la cantidad de notificaciones e información de depuración que se imprimirá en la consola. `0` es sin salida, `1` es salida normal, y `2` es salida con explicaciones.

G.3.10. --adminmedia

Ejemplo de uso:

```
django-admin.py --adminmedia=/tmp/new-admin-style/
```

Le dice a Django donde encontrar los archivos CSS y JavaScript para la interfaz de administración cuando se ejecuta el servidor de desarrollo. Normalmente estos archivos son servidos por fuera del arbol de fuentes Django pero como algunos diseñadores personalizan estos archivos para su sitio, esta opción te permite testear con versiones personalizadas.

Apéndice H

Objetos Petición y Respuesta

Django usa los objetos respuesta y petición 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` y `HttpResponse`.

H.1. HttpRequest

`HttpRequest` representa una sola petición HTTP desde algún agente de usuario.

Mucha de la información importante sobre la petición esta disponible como atributos en la instancia de `HttpRequest` (mira la Tabla H-1). Todos los atributos excepto `session` deben considerarse de sólo lectura.

Cuadro H.1: Atributos de los objetos HttpRequest

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()`

`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 más sobre el uso de cookies.

Cuadro H.1: Atributos de los objetos HttpRequest

Atributo	Descripción
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 esta disponible en META como claves con el prefijo HTTP_, por ejemplo: ▪ HTTP_ACCEPT_ENCODING ▪ HTTP_ACCEPT_LANGUAGE ▪ HTTP_HOST: La cabecera HTTP host enviada 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 esta establecida.

user	Un objeto django.contrib.auth.models.User que representa el usuario actual registrado. Si el usuario no esta 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():`
	`  # Do something for logged-in users.`
	`else:`
	`  # Do something for anonymous users.`
	user esta disponible sólo si tu instalación Django tiene activado AuthenticationMiddleware.
	Para los detalles completos sobre autenticación y usuarios, mira el Capítulo 12.

session	Un objeto similar a un diccionario que se puede leer y modificar, que representa la sesión actual. Éste esta disponible sólo si tu instalación Django tiene activado el soporte para sesiones. Mira el Capítulo 12.

Cuadro H.1: Atributos de los objetos HttpRequest

Atributo	Descripción
raw_post_data	Los datos HTTP POST en crudo. Esto es útil para procesamiento avanzado.

Los objetos request también tienen algunos métodos de utilidad, como se muestra en la Tabla H-2.

Cuadro H.2: Métodos de HttpRequest

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_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.

H.1.1. 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 tu 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 H-3.

Cuadro H.3: Como se diferencian los QueryDicts de los diccionarios estándar.

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']

Cuadro H.3: Como se diferencian los QueryDicts de los diccionarios estándar.

Método	Diferencias con la implementación estándar de dict

`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()__`.

Además, `QueryDict` posee los métodos que se muestran en la Tabla H-4.

Cuadro H.4: Métodos QueryDict Extra (No relacionados con diccionarios)

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"`).

H.1.2. Un ejemplo completo

Por ejemplo, dado este formulario HTML:

```
<form action="/foo/bar/" method="post">
<input type="text" name="your_name" />
<select multiple="multiple" name="bands">
    <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 `your_name` 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
{'your_name': ['John Smith'], 'bands': ['beatles', 'zombies']}
```

```
>>> request.POST['your_name']
'John Smith'
>>> request.POST['bands']
'zombies'
>>> request.POST.getlist('bands')
['beatles', 'zombies']
>>> request.POST.get('your_name', 'Adrian')
'John Smith'
>>> request.POST.get('nonexistent_field', 'Nowhere Man')
'Nowhere Man'
```

Nota de implementación:

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.

H.2. 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` esta ubicada en `django.http.HttpResponse`.

H.2.1. Construcción de HttpResponses

Típicamente, tu construirás un `HttpResponse` para pasar los contenidos de la pagina, como un string, al constructor de `HttpResponse`:

```
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", 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>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</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 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 el Capítulo 11 para ver algunos ejemplos de la utilización de esta técnica.

H.2.2. Establecer las cabeceras

Puedes agregar o eliminar cabeceras usando sintaxis de diccionario:

```
>>> response = HttpResponse()
>>> response['X-DJANGO'] = "It's the best."
>>> del response['X-PHP']
>>> response['X-DJANGO']
"It's the best."
```

Puedes utilizar también `has_header(header)` para verificar la existencia de una cabecera.

Evita configurar cabeceras `Cookie` a mano; en cambio, mira el Capítulo 12 para instrucciones sobre como trabajan las cookies en Django.

H.2.3. Subclases de HttpResponse

Django incluye un numero de subclases `HttpResponse` que manejan diferentes tipos de respuestas HTTP (mira la Tabla H-5). Así como `HttpResponse`, estas subclases se encuentran en `django.http`.

Cuadro H.5: Subclasses de HttpResponse

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

Puedes, por supuesto, definir tus propias subclases de `HttpResponse` para permitir diferentes tipos de respuestas no admitidas por las clases estándar.

H.2.4. 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 my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</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>Page not found</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 tu emites una `Http404` en cualquier punto de una vista de 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 detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.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.

H.2.5. 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 tu quieres reemplazar la vista 404, puedes especificar `handler404` en tu URLconf, de la siguiente manera:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    ...
)

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.defaults import *
```

Esto se encarga de establecer `handler404` en el módulo actual. Como puedes ver en `django/conf/urls/defaults.py`, `handler404` esta fijado a `'django.views.defaults.page_not_found'` por omisión.

Hay tres cosas para tener en cuenta sobre las vistas 404:

- La vista 404 es llamada también si Django no encuentra una coincidencia después de verificar toda expresión regular en la URLconf.
- Si no defines tu propia vista 404 -- y simplemente usas la predeterminada, lo cual es recomendado -- tu 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.
- Si `DEBUG` esta 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.

H.2.6. Personalizar la Vista 500 (Server Error)

De manera similar, Django ejecuta comportamiento de caso 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 tu quieres reemplazar la vista, puedes especificar `handler500` en tu URLconf, de la siguiente manera:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    ...
)

handler500 = 'mysite.views.my_custom_error_view'
```

Apéndice I

Docutils System Messages

Indirect hyperlink target ”capítulo 10“ (id=”cap-tulo-10“) refers to target ”extendiendo el sistema de plantillas“, which is a duplicate, and cannot be used as a unique reference.

Indirect hyperlink target ”capítulo 19“ (id=”cap-tulo-19“) refers to target ”seguridad“, which is a duplicate, and cannot be used as a unique reference.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Unknown target name: ”empaquetado de funciones vista“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Unknown target name: ”contraseñas“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Duplicate target name, cannot be used as a unique reference: ”próximo capítulo“.

Unknown target name: ”nombres de tabla“.

Generated on: 2008-07-28 23:53 UTC. Generated by Docutils from reStructuredText source.