Tutorial de Django

Release 1.5

Django Software Foundation

February 23, 2013

Índice general

CAPÍTULO 1

Empezando con Django

Nuevo en Django? O en desarrollo web en general? Bueno, estás en el lugar indicado: leé este material para empezar rápidamente.

1.1 Django de un vistazo

Como Django fue desarrollado en el entorno de una redacción de noticias, fue diseñado para hacer las tareas comunes del desarrollo web rápidas y fáciles. Esta es una introducción informal de cómo escribir una aplicación basada en una base de datos con Django.

El objetivo de este documento es brindar las especificaciones técnicas suficientes para entender cómo funciona Django, pero no ser un tutorial o una referencia – ambos existen! Cuando estés listo para empezar un proyecto, podés *chequear el tutorial* o sumergirte en la `documentación más detallada`.

1.1.1 Diseñar tu modelo

Aunque se puede usar sin una base de datos, Django viene con un mapeador objeto-relacional a través del cual podés describir la estructura de tu base de datos en código Python.

La `sintaxis de modelos de datos` ofrece muchas maneras de representar tus modelos – hasta el momento, ha resuelto problemas de base de datos durante años. Aquí hay un ejemplo rápido, que podrías guardar en el archivo `mysite/news/models.py`:

```
class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __unicode__(self):
        return self.full_name

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)

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

1.1.2 Instalarlo

A continuación hay que correr la utilidad de línea de comandos de Django para crear las tablas de la base de datos automáticamente:

```
manage.py syncdb
```

El comando `syncdb` revisa todos tus modelos disponibles y crea tablas en la base de datos para aquellos modelos cuyas tablas todavía no existen.

1.1.3 Aprovecha la API ya provista

En este punto ya obtenés una completa `API Python` para acceder a tus datos. La API es creada "al vuelo", sin requerir generación de código:

```
>>> from news.models import Reporter, Article
# Importamos los modelos creados en nuestra app "news"

# Todavía no hay periodistas en el sistema.
>>> Reporter.objects.all()
[]

# Creamos un nuevo periodista.
>>> r = Reporter(full_name='John Smith')

# Guardamos el objeto en la base de datos. Hay que llamar save() explícitamente.
>>> r.save()

# Ahora el objeto tiene un ID.
>>> r.id
1

# Ahora el periodista está en la base de datos.
>>> Reporter.objects.all()
[<Reporter: John Smith>]

# Los campos se representan como atributos del objeto Python.
>>> r.full_name
'John Smith'

# Django provee una completa API para realizar búsquedas.
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist.

# Creamos un artículo.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool',
...     content='Yeah.', reporter=r)
>>> a.save()
```

```
# Ahora el artículo está en la base de datos.
>>> Article.objects.all()
[<Article: Django is cool>]

# Los objetos Article tienen acceso a los objetos Reporter relacionados.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# Y vice versa: los objetos Reporter tienen acceso a los objetos Article.
>>> r.article_set.all()
[<Article: Django is cool>]

# La API sigue las relaciones tan lejos como necesites,
# haciendo eficientes JOINs detrás de escena.
# Esto busca todos los artículos de un periodista cuyo nombre empieza con "John".
>>> Article.objects.filter(reporter__full_name__startswith="John")
[<Article: Django is cool>]

# Cambiamos un objeto modificando sus atributos y llamando a save().
>>> r.full_name = 'Billy Goat'
>>> r.save()

# Borramos un objeto con delete().
>>> r.delete()
```

1.1.4 Interfaz de administración dinámica: no sólo los andamios – la casa completa

Una vez que tus modelos están definidos, Django puede crear automáticamente una `interfaz de administración` profesional, lista para producción – un sitio web que permite a usuarios autenticados agregar, modificar y borrar objetos. Es tan fácil como registrar tu modelo en el sitio de administración:

```
# En el archivo models.py...

from django.db import models

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)


# En el archivo admin.py en el mismo directorio...

import models
from django.contrib import admin

admin.site.register(models.Article)
```

La filosofía aquí es que tu sitio es editado por un staff, un cliente o quizás solamente vos – y vos no querés tener que crear las interfaces de backend solamente para manejar el contenido.

Un flujo típico al crear las apps Django es definir los modelos y configurar el sitio de administración corriendo tan rápido como sea posible, de tal forma que el staff (o los clientes) pueden empezar a agregar información. Y luego, desarrollar la manera en que esta información es presentada al público.

1.1.5 Diseñar tus URLs

Un esquema de URLs limpio y elegante es un detalle importante en una aplicación web de calidad. Django incentiva el diseño elegante de URLs y no añade ningún sufijo como `.php` or `.asp`.

Para diseñar las URLs de una app, hay que crear un módulo Python llamado `URLconf`. Es una tabla de contenidos para tu app, contiene un simple mapeo entre patrones de URL y funciones Python. Los URLconfs también sirven para desacoplar las URLs del código Python.

A continuación cómo podría ser un URLconf para el ejemplo anterior de `Reporter/Article`:

```
from django.conf.urls import patterns

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

El código de arriba mapea URLs, listadas como expresiones regulares simples, a la ubicación de funciones Python de callback ("views"). Las expresiones regulares usan paréntesis para "capturar" valores en las URLs. Cuando un usuario pide una página, Django recorre los patrones, en orden, y se detiene en el primero que coincide con la URL solicitada. (Si ninguno coincide, Django llama a un view especial para un 404.) Esto es muy rápido porque las expresiones regulares se compilan cuando se carga el código.

Una vez que una de las expresiones regulares coincide, Django importa e invoca la view correspondiente, que es simplemente una función Python. Cada view recibe como argumentos un objeto request – que contiene la metada del request – y los valores capturados en la expresión regular.

Por ejemplo, si el usuario solicita la URL "/articles/2005/05/39323/", Django llamaría a la función `news.views.article_detail(request, '2005', '05', '39323')`.

1.1.6 Escribir tus views

Cada view es responsable de hacer una de dos cosas: devolver un objeto `HttpResponse` con el contenido de la página solicitada, o levantar una excepción como `Http404`. El resto depende de cada uno.

Generalmente, una view obtiene datos de acuerdo a los parámetros que recibe, carga un template y lo renderiza usando esos datos. Este es un ejemplo de una view para `year_archive` siguiendo con el ejemplo anterior:

```
def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    return render_to_response('news/year_archive.html', {'year': year, 'article_list': a_list})
```

Este ejemplo usa el `sistema de templates` de Django, que tiene varias características poderosas pero es lo suficientemente simple de usar para no-programadores.

1.1.7 Diseñar tus templates

El código anterior cargar el template `news/year_archive.html`.

Django tiene un path para la búsqueda de templates, que te permite minimizar la redundancia entre templates. En los settings de tu proyecto Django podés especificar una lista de directorios en los cuales buscar templates. Si un template no existe en el primer directorio, se chequea el segundo, y así sucesivamente.

Supongamos que el template `news/year_archive.html` se encuentra, su contenido podría ser:

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

{% block title %}Articles for {{ year }}{% endblock %}

{% block content %}
<h1>Articles for {{ year }}</h1>

{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

Las variables se encierran entre doble llaves. `{{ article.headline }}` significa "Escribir el valor del atributo headline del objeto article." Pero los puntos no solamente se usan para atributos: también se usan para acceder a una clave de un diccionario, acceder a un índice y llamadas a función.

Notar que `{{ article.pub_date|date:"F j, Y"}}` usa un "pipe" (el caracter "|") al estilo Unix. Se trata de lo que se llama un template filter, y es una manera de aplicar un filtro al valor de una variable. En este caso, el filtro date formatea un objeto datetime de Python con el formato dado (como en el caso de la función date de PHP).

Podés encadenar tantos filtros como quieras. También podés escribir filtros propios. Podés escribir template tags personalizados, que corren código Python propio detrás de escena.

Finalmente, Django usa el concepto de "herencia de templates": eso es lo que hace `{% extends "base.html" %}`. Significa "Primero cargar el template llamado 'base', que define una serie de bloques, y rellenar esos bloques con los siguientes bloques". En síntesis, permite recortar drásticamente la redundancia en los templates: cada template solamente tiene que definir lo que es único para él mismo.

El template "base.html" podría ser algo como:

```
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="sitelogo.png" alt="Logo" />
    {% block content %}{% endblock %}
</body>
</html>
```

Simplificando, define el look-and-feel del sitio (con el logo del sitio) y provee los "espacios" para que los templates hijos completen. Esto hace que el rediseño de un sitio sea tan sencillo como modificar un único archivo – el template base.

También permite crear múltiple versiones de un sitio, con diferentes templates base y reusando los templates hijos. Los creadores de Django han usado esta técnica para crear completamente diferentes de sitios para su versión móvil – simplemente creando un nuevo template base.

Hay que notar que si uno prefiere puede usar un sistema de templates distinto al de Django. Si bien el sistema de templates de Django está particularmente bien integrado con la capa de Django de modelos, nada nos fuerza a usarlo. Tampoco es obligatorio usar la capa de abstracción de la base de datos provista por Django. Se puede usar otra abstracción, leer de archivos XML, leer de archivos de disco, o lo que uno quiera. Cada pieza de Django – modelos, views, templates – está desacoplada del resto.

1.1.8 Esto es sólo la superficie

Esta es tan sólo un rápido vistazo a la funcionalidad de Django. Algunas otras características útiles:

- Un `framework de caching` que se integra con memcached y otros backends.
- Un `framework de sindicación` que hace que crear feeds RSS y Atom sea tan fácil como escribir una pequeña clase Python.
- Más y mejores características en la creación automática del sitio de administración – esta introducción apenas trata el tema superficialmente.

Los siguientes pasos obvios son bajar Django, leer el *tutorial* y unirse a la comunidad. Gracias por tu interés!

1.2 Guía de instalación rápida

Antes de poder usar Django es necesario instalarlo. Existe una `guía de instalación completa` que cubre todas las posibilidades; esta guía es simple, cubre una instalación mínima que servirá mientras se recorre la introducción.

1.2.1 Instalar Python

Siendo un framework web escrito en Python, Django requiere Python. Funciona con cualquier versión de Python desde 2.6.5 hasta 2.7. También existe soporte experimental para 3.2 y 3.3. Estas versiones de Python incluyen una base de datos liviana llamada SQLite, así que no es necesario configurar un motor de base datos inmediatamente.

1.2.2 Instalar Django

Siendo un framework web escrito en Python, Django requiere Python. Funciona con cualquier versión de Python desde 2.5 hasta 2.7 (debido a incompatibilidades para atrás en Python 3.0, Django no funciona actualmente con Python 3.0; para mayor información sobre las versiones de Python soportadas y la transición a 3.0, podés ver la `Django FAQ`); estas versiones de Python incluyen una base de datos liviana llamada SQLite, así que no es necesario configurar un motor de base datos inmediatamente.

Podés conseguir Python en http://www.python.org. Si estás corriendo Linux o Mac OS X, probablemente ya lo tengas instalado.

Django en Jython

Si usás Jython (implementación de Python para la plataforma Java), es necesario seguir algunas pasos adicionales. Ver detalles en `/howto/jython`.

Podés verificar que Python está instalado corriendo `python` en tu shell; deberías ver algo como:

```
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.2.3 Configurar una base de datos

Si estás usando Python 2.5 o superior, podés saltear este paso por ahora.

Si no, o si quisieras trabajar con un motor de base de datos más "grande" como PostgreSQL, MySQL u Oracle, consultá la *información sobre la instalación de base de datos*.

1.2.4 Borrar versiones anteriores de Django

Si estás actualizando una versión previa de Django es necesario *desinstalar la versión anterior antes de instalar una nueva*.

1.2.5 Instalar Django

Existen tres opciones fáciles de instalar Django:

- Instalar una versión de Django `provista por tu sistema operativo`. Esta es la opción más rápida para aquellos que tienen un sistema operativo que distribuye Django.
- *Instalar un release oficial*. Esta es la mejor alternativa para los usuarios qie quieren un número de versión estable y que no les preocupa no correr la versión más reciente de Django.
- *Instalar la última versión de desarrollo*. Esta es la mejor opción para los usuarios que quieren las características más recientes y que no tienen miedo de correr código nuevo.

Siempre recurrir a la versión de la documentación que corresponde a la versión de Django que estás usando!

Si seguís cualquiera de los dos primeros pasos, hay que estar atentos a las partes de la documentación marcadas como **new in development version** (nuevo en la versión de desarrollo). Esta frase indica características que están solamente disponibles en la versión de desarrollo de Django, y que muy posibiblemente no funcionen en un release oficial.

1.2.6 Verificando

Para verificar que Django es accesible desde Python, tipeá `python` en tu shell. Una vez en el prompt de Python, intentá importar Django:

```
>>> import django
>>> print(django.get_version())
1.5
```

You may have another version of Django installed.

1.2.7 Eso es todo!

Así es – ahora podés *seguir con el tutorial*.

1.3 Escribiendo tu primera Django app, parte 1

Vamos a aprender mediante un ejemplo.

A lo largo de este tutorial vamos a recorrer la creación de una aplicación de encuestas básica.

Consistirá de dos partes:

- Un sitio público que permite a la gente ver y votar encuestas.
- Un sitio de administración que nos permite agregar, cambiar y borrar encuestas.

Vamos a asumir que tenés *Django ya instalado*. Podés chequear esto, así como también la versión, corriendo el siguiente comando:

```
python -c "import django; print(django.get_version())"
```

Deberías o bien ver la versión de Django instalado o un error diciendo "No module django". Podés confirmar que la versión coincide con la de este tutorial. En caso de no coincidir, podés buscar el tutorial para tu versión, o actualizar Django.

Acá `How to install Django` encontrarás cómo borrar versiones anteriores de Django e instalar una más reciente.

Dónde obtener ayuda:

Si estás teniendo problemas siguiendo este tutorial, por favor posteá un mensaje a django-users o en #django on irc.freenode.net para chatear con otros usuarios de Django que quizás puedan ayudar.

1.3.1 Creando un proyecto

Si esta es tu primera vez usando Django, tenés que hacer un setup inicial. En particular, necesitás auto-generar algo de código que define un Django *project* – una colección de settings para una instancia de Django, que incluye la configuración de la base de datos, opciones específicas de Django y settings específicos de las aplicaciones.

Desde la línea de comandos, `cd` al directorio donde quisieras guardar tu código, y corré el siguiente comando:

```
django-admin.py startproject mysite
```

Esto creará el directorio `mysite` en tu directorio actual. Si no funcionó, podés ver *Problemas corriendo django-admin.py*.

Nota: Hay que evitar nombrar los proyectos que coincidan con componentes built-in de Python o Django. En particular, significa que uno no debería usar nombres tales como `django` (en conflicto con Django mismo) o `test` (en conflicto con el paquete built-in `test` de Python).

Dónde debería estar este código?

Si tu background es en PHP (sin usar un framework moderno), probablemente estés acostumbrado a poner el código en la raíz del servidor web (un lugar como `/var/www`). Con Django no se hace así. No es una buena idea poner código Python en dicho lugar, porque existe el riesgo de que la gente pueda ver tu código en la web. Eso no es bueno en relación a la seguridad.

Uno pone el código en algún directorio **fuera** de la raíz del servidor web, como `/home/mycode`.

Veamos lo que creó `startproject`:

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

No es lo que estás viendo?

La estructura por defecto de un proyecto ha cambiado recientemente. Si estás viendo una estructura "plana" (sin un directorio interno `mysite/`), probablemente estés usando una versión de Django que no coincide con la de este tutorial. Quizás quieras cambiar a una versión anterior del tutorial o a una versión más nueva de Django.

Estos archivos son:

- El directorio `mysite/` de más afuera es sólo un contenedor para tu proyecto. El nombre no afecta a Django; lo podés renombrar libremente como quieras.
- `manage.py`: Una utilidad de línea de comandos que te permite interactuar con este proyecto Django de varias maneras. Podés leer todos los detalles sobre `manage.py` en `/ref/django-admin`.
- El directorio `mysite/` interno es el paquete Python para tu proyecto. Su nombre es el nombre de paquete Python que necesitarás usar para importar cualquier cosa adentro del mismo (e.g. `import mysite.settings`).
- `mysite/__init__.py`: Un archivo vacío que le dice a Python que este directorio debe considerarse un paquete Python (si sos nuevo con Python, podés leer más sobre paquetes en la documentación oficial de Python).
- `mysite/settings.py`: Settings/configuración de este proyecto Django. `/topics/settings` describe cómo funcionan estos settings.
- `mysite/urls.py`: Declaración de las URL de este proyecto Django; una "tabla de contenidos" de tu sitio Django. Podés leer más sobre URLs en `/topics/http/urls`.
- `mysite/wsgi.py`: Punto de entrada para servir tu proyecto mediante servidores web compatibles con WSGI. Podés ver `/howto/deployment/wsgi/index` para más detalles.

El servidor de desarrollo

Veamos que esto funcionó. Nos cambiamos al directorio `mysite` de más afuera, si no lo habíamos hecho ya, y corremos el comando `python manage.py runserver`. Deberíamos obtener la siguiente salida en la terminal:

```
Validating models...

0 errors found
February 17, 2013 - 15:50:53
Django version 1.5, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Hemos levantado el servidor de desarrollo de Django, un servidor web liviano escrito puramente en Python. Viene incluido con Django para permitir desarrollar rápidamente, sin necesidad de configurar un servidor de producción – como Apache – hasta el momento en que todo esté listo para producción.

Este es un bien momento para notar: NO hay que usar este servidor para nada que se parezca a un entorno de producción. Está pensado solamente para desarrollo (Django es un framework web, no un servidor).

Ahora que el servidor está corriendo, podemos visitar http://127.0.0.1:8000/ en nuestro browser. Deberíamos ver una página con el mensaje "Welcome to Django". Funcionó!

Cambiando el puerto

Por defecto, el comando `runserver` levanta el servidor de desarrollo en una IP interna en el puerto 8000.

Si uno quisiera cambiar el puerto, se puede pasar como argumento en la línea de comandos. Por ejemplo, para levantar el servidor escuchando en el puerto 8080:

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

Para cambiar la dirección IP del servidor, se pasa junto con el puerto. Entonces, para escuchar en todas las IP públicas (útil para mostrarle nuestro trabajo en otras computadoras), podemos usar:

```
python manage.py runserver 0.0.0.0:8000
```

La documentación completa sobre el servidor de desarrollo se puede encontrar en `runserver`.

Configuración de la base de datos

Ahora editamos `mysite/settings.py`. Es un módulo Python normal con variables a nivel módulo que representan los parámetros de configuración de Django. Cambiamos los siguientes valores de `'default'` en el `DATABASES` de acuerdo a los parámetros de conexión a nuestra base de datos.

- `ENGINE` – Puede ser `'django.db.backends.postgresql_psycopg2'`, `'django.db.backends.mysql'`, `'django.db.backends.sqlite3'` o `'django.db.backends.oracle'`. También hay `otros backends disponibles`.
- `NAME` – El nombre de la base de datos. Si estás usando SQLite, tu base de datos será un archivo en tu computadora; en ese caso, `NAME` debería ser un path absoluto, incluyendo el nombre del archivo de base de datos. Si no existiera, se creará automáticamente cuando se sincronice la base de datos por primera vez (ver más abajo).

  Cuando se especifique un path, siempre usar barras, incluso en Windows (e.g. `C:/homes/user/mysite/sqlite3.db`).
- `USER` – Nombre de usuario de la base de datos (no se usa para SQLite).
- `PASSWORD` – El password de la base de datos (no se usa para SQLite).
- `HOST` – El host en el que está la base de datos. Dejarlo vacío (o posiblemente `127.0.0.1`) si el servidor de base de datos está en la misma máquina física (no se usa para SQLite). Ver `HOST` para más detalles.

Si sos nuevo con base de datos, recomendamos simplemente usar SQLite seteando `ENGINE` a `'django.db.backends.sqlite3'` y `NAME` al lugar en el que quieras guardar la base de datos. SQLite está incluido en Python, no hace falta instalar nada más.

Nota: Si usás PostgreSQL o MySQL, fijate de crear una base de datos antes de seguir. Para ello bastará con hacer "`CREATE DATABASE database_name;`" en el intérprete del motor correspondiente.

Si usás SQLite, no es necesario crear nada de antemano - el archivo de la base de datos se creará automáticamente cuando haga falta.

Mientras editás `settings.py`, podés setear `TIME_ZONE` a tu zona horaria. El valor por defecto es el huso horario Central de Estados Unidos (Chicago).

También podés mirar el setting `INSTALLED_APPS` hacia el final del archivo. Éste registra los nombres de todas las aplicaciones Django que están activadas en esta instancia Django. Las apps se pueden usar en múltiples proyectos, y podés empaquetarlas y distribuirlas para su uso por otros en sus respectivos proyectos.

Por defecto, `INSTALLED_APPS` contiene las siguientes apps, todas provistas por Django:

- `django.contrib.auth` – Sistema de autenticación.
- `django.contrib.contenttypes` – Un framework para tipos de contenido.
- `django.contrib.sessions` – Un framework para manejo de sesiones.
- `django.contrib.sites` – Un framework para la administración de múltiples sitios con una instalación de Django.
- `django.contrib.messages` – Un framework de mensajes.

- `django.contrib.staticfiles` – Un framework para manejar los archivos estáticos.

Estas aplicaciones están incluidas por defecto como conveniencia para el caso común.

Cada una de estas aplicaciones hace uso de al menos una tabla de la base de datos, entonces necesitaremos crear las respectivas tablas antes de poder usarlas. Para ello corremos el siguiente comando:

```
python manage.py syncdb
```

El comando `syncdb` se fija en el setting `INSTALLED_APPS` y crea las tablas necesarias en la base de datos determinada por los parámetros establecidos en el archivo `settings.py`. Verás un mensaje por cada tabla que se crea, y tendrás la opción de crear una cuenta de superusuario para el sistema de autenticación. Hagámoslo.

Si te interesa, podés correr el cliente de línea de comandos de tu motor de base de datos y hacer `\dt` (PostgreSQL), `SHOW TABLES;` (MySQL), o `.schema` (SQLite) para ver las tablas creadas por Django.

Para los minimalistas

Como dijimos arriba, las aplicaciones incluidas por defecto son para el caso común, pero no todos las necesitan. Si no necesitás alguna o ninguna de las mismas, sos libre de comentar o borrar las líneas apropiadas de `INSTALLED_APPS` antes de correr `syncdb`. El comando `syncdb` sólo creará las tablas para las apps en `INSTALLED_APPS`.

1.3.2 Creando modelos

Ahora que hemos levantado nuestro entorno – un "proyecto" –, estamos listos para empezar a trabajar.

Cada aplicación que uno escribe en Django consiste de un paquete Python, en algún lugar del Python path, siguiendo ciertas convenciones. Django trae una utilidad que automáticamente genera la estructura de directorios básica de una app, de tal manera que uno pueda concentrarse en escribir código en lugar de directorios.

Proyectos vs. apps

Cuál es la diferencia entre un proyecto y una app? Una app es una aplicación web que hace algo – e.g., un sistema de blog, una base de datos de registros públicos o una aplicación simple de encuestas. Un proyecto es una colección de configuración y apps para un sitio web particular. Un proyecto puede contener múltiples app. Una app puede estar en múltiples proyectos.

Las apps viven en cualquier lugar del Python path. En este tutorial, vamos a crear nuestra app en el directorio donde se encuentra el archivo `manage.py`, para que pueda ser importada como módulo de primer nivel, en lugar de ser un submódulo de `mysite`.

Para crear una app, nos aseguramos de estar en el mismo directorio que `manage.py` y corremos el comando:

```
python manage.py startapp polls
```

Esto creará el directorio `polls`, con la siguiente estructura:

```
polls/
    __init__.py
    models.py
    tests.py
    views.py
```

Esta estructura de directorio va a almacenar la aplicación `poll`.

El primer paso al escribir una app web en Django es definir los modelos – esencialmente, el esquema de base de datos, con metadata adicional.

Filosofía

Un modelo es la única y definitiva fuente de datos de nuestra información. Contiene los campos y comportamientos esenciales de los datos que vamos a guardar. Django sigue el *principio DRY*. El objetivo es definir el modelo de datos en un lugar y automáticamente derivar lo demás a partir de éste.

En nuestra simple app poll, vamos a crear dos modelos: `Poll` and `Choice`. Una `Poll` tiene una pregunta y una fecha de publicación. Una `Choice` tiene dos campos: el texto de la opción y un contador de votos. Cada `Choice` está asociadaa una `Poll`.

Estos conceptos se representan mediante clases Python. Editamos el archivo `polls/models.py` para que se vea así:

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField()
```

El código es directo. Cada modelo se representa por una clase que hereda de `django.db.models.Model`. Cada modelo tiene ciertas variables de clase, cada una de las cuales representa un campo de la base de datos en el modelo.

Cada campo se representa como una instancia de una clase `Field` – e.g., `CharField` para campos de caracteres y `DateTimeField` para fecha y hora. Esto le dice a Django qué tipo de datos almacena cada campo.

El nombre de cada instancia de `Field` (e.g. `question` o `pub_date`) es el nombre del campo, en formato amigable (a nivel código). Vamos a usar este valor en nuestro código, y la base de datos lo va a usar como nombre de columna.

Se puede usar un primer argumento, opcional, de `Field` para designar un nombre legible (a nivel ser humano). Se usa en algunas partes en que Django hace introspección, y funciona como documentación. Si no se provee este argumento, Django usa el nombre del campo. En el ejemplo, solamente definimos un nombre descriptivo para `Poll.pub_date`. Para todos los demás campos en el modelo, el nombre del campo será suficiente.

Algunas clases de `Field` tienen argumentos requeridos. Por ejemplo, `CharField` requiere que se pase `max_length`. Esto se usa no sólo en el esquema de la base de datos sino también en la validación de los datos, como veremos más adelante.

Finalmente, notemos que se define una relación, usando `ForeignKey`. Esto le dice a Django que cada `Choice` está relacionada a una única `Poll`. Django soporta todos los tipos de relación comunes en una base de datos: muchos-a-uno, muchos-a-muchos y uno-a-uno.

1.3.3 Activando modelos

Ese poquito código le da a Django un montón de información. A partir de él, Django puede:

- Crear el esquema de base de datos (las sentencias `CREATE TABLE`) para la app.
- Crear una API Python de acceso a la base de datos para los objetos `Poll` y `Choice`.

Pero primero debemos informarle a nuestro proyecto que la app `polls` está instalada.

Filosofía

Las apps Django son "pluggable": podés usar una app en múltiples proyectos, y distribuirlas, porque no necesitan estar ligadas a una instancia de Django particular.

Editamos de nuevo el archivo `settings.py`, y cambiamos el setting `INSTALLED_APPS` para incluir `'polls'`. Se verá algó así:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
)
```

Ahora Django sabe sobre nuestra app `polls`. Corramos otro comando:

```
python manage.py sql polls
```

Deberíamos ver algo similar a lo siguiente (las sentencias SQL `CREATE TABLE` de la app polls):

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id") DEFERRABLE INITIALLY DEFERRED,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
COMMIT;
```

Notemos lo siguiente:

- La salida exacta varía de acuerdo a la base de datos que se esté usando.
- Los nombres de las tablas se generan automáticamente combinando el nombre de la app (`polls`) con el nombre, en minúsculas del modelo – `poll` y `choice` (se puede modificar este comportamientos).
- Las claves primarias (IDs) se agregan automáticamente (esto también se puede modificar).
- Por convención, Django añade `"_id"` al nombre del campo de clave foránea (sí, se puede modificar esto también).
- La relación de clave foránea se hace explícita mediante la sentencia `REFERENCES`.
- Se ajusta a la base de datos que se esté usando, y entonces los tipos de campos específicos de la base de datos como `auto_increment` (MySQL), `serial` (PostgreSQL), o `integer primary key` (SQLite) se manejan por uno automáticamente. Lo mismo aplica para los nombres de los campos – e.g., el uso de comillas dobles o simples. El autor de este tutorial corre PostgreSQL, la salida del ejemplo por lo tanto refleja la sintaxis de PostgreSQL.

- El comando `sql` no corre el SQL en la base de datos - solamente imprime por pantalla el SQL que Django piensa que es requerido. Si quisieras, podrías copiar y pegar el SQL en el prompt de tu base de datos. Sin embargo, como veremos en breve, Django provee una forma más fácil de ejecutar este SQL en la base de datos.

Si estás interesado, podés correr también estos comandos:

- `python manage.py validate` – Chequea por errores en la definición de los modelos.
- `python manage.py sqlcustom polls` – Imprime *sentencias SQL personalizadas* (como modificaciones de tablas o restricciones) definidas por la aplicación.
- `python manage.py sqlclear polls` – Imprime las sentencias `DROP TABLE` necesarias para esta aplicación, de acuerdo a las tablas existentes en la base de datos (si hubiera).
- `python manage.py sqlindexes polls` – Imprime las sentencias `CREATE INDEX` para esta aplicación.
- `python manage.py sqlall polls` – Una combinación de los comandos `sql`, `sqlcustom`, y `sqlindexes`.

Mirando la salida de estos comandos puede ayudar a entender qué es lo que sucede realmente por detrás.

Ahora, corremos `syncdb` de nuevo para crear las tablas de los modelos en la base de datos:

```
python manage.py syncdb
```

El comando `syncdb` corre el SQL de `sqlall` en la base de datos para todas las apps en `INSTALLED_APPS` que no existan previamente en la base de datos. Esto crea todas las tablas, datos iniciales e índices para cualquier app que hayamos agregado a nuestro proyecto desde la última vez que corrimos syncdb. `syncdb` se puede ejecutar tan frecuentemente como querramos, y sólo va a crear las tablas que no existan previamente.

Para tener la información completa de qué puede hacer la utilidad `manage.py`, podés leer la documentación de `django-admin.py`.

1.3.4 Jugando con la API

Ahora pasemos al intérprete interactivo de Python y juguemos con la API que Django nos provee. Para invocar el shell de Python, usamos este comando:

```
python manage.py shell
```

Usamos esto en lugar de simplemente tipear "python" porque `manage.py` setea la variable de entorno `DJANGO_SETTINGS_MODULE`, que le da a Django el import path al archivo `settings.py`.

Bypassing manage.py

Si preferís no usar `manage.py`, no hay problema. Basta setear la variable de entorno `DJANGO_SETTINGS_MODULE` a `mysite.settings` y correr `python` desde el mismo directorio en que está `manage.py` (o asegurarse que este directorio está en el Python path, para que `import mysite` funcione).

Para más información sobre todo esto, podés ver `django-admin.py`.

Una vez en el shell, exploramos la `API de base de datos`:

```
>>> from polls.models import Poll, Choice
# Importamos las clases de los modelos que escribimos.

# No hay encuestas en el sistema todavía.
>>> Poll.objects.all()
[]
```

```
# Creamos una nueva encuesta.
# El soporte para zonas horarias está habilitado por defecto en settings.py
# Django espera entonces un datetime con tzinfo para pub_date.
# Usamos timezone.now() en lugar de datetime.datetime.now(), y
# esto hace lo correcto.
>>> from django.utils import timezone
>>> p = Poll(question="What's new?", pub_date=timezone.now())

# Guardamos el objeto en la base de datos.
# Debemos llamar a save() explícitamente.
>>> p.save()

# Ahora tiene un ID. Notar que esto puede decir "1L" en lugar de "1",
# dependiendo de la base de datos que se esté usando. Esto sólo significa
# que el backend del motor de base de datos prefiere devolver los enteros
# como long integer de Python.
>>> p.id
1

# Podemos acceder a las columnas de la base de datos como atributos de Python.
>>> p.question
"What's new?"
>>> p.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Podemos cambiar valores cambiando el valor de los atributos.
# Luego llamamos a save().
>>> p.question = "What's up?"
>>> p.save()

# objects.all() muestra todas las encuestas en la base de datos.
>>> Poll.objects.all()
[<Poll: Poll object>]
```

Un minuto. `<Poll: Poll object>` es, definitivamente, una representación poco útil de este objeto. Arreglemos esto editando los modelos (en el archivo `polls/models.py`) y agregando el método `__unicode__()` a `Poll` and `Choice`:

```
class Poll(models.Model):
    # ...
    def __unicode__(self):
        return self.question

class Choice(models.Model):
    # ...
    def __unicode__(self):
        return self.choice_text
```

Es importante agregar el método `__unicode__()` a nuestros modelos, no sólo por nuestra salud al tratar con el intérprete, sino también porque es la representación usada por Django en la interfaz de administración autogenerada.

Por qué `__unicode__()` y no `__str__()`?

Si estás familiarizado con Python, puede que tengas el hábito de agregar métodos `__str__()` a tus clases, no `__unicode__()`. Aquí usamos `__unicode__()` porque los modelos de Django manejan Unicode por defecto. Todos los datos guardados en la base de datos se convierten a Unicode cuando se leen.

Los modelos de Django tienen un método `__str__()` por defecto que llama a `__unicode__()` y convierte el

resultado a un bytestring UTF-8. Esto significa que `unicode(p)` devolverá un string Unicode, y `str(p)` un string normal con los caracteres encodeados en UTF-8.

Si todo esto te es mucho ruido, solamente recordá agregar un método `__unicode__()` a tus modelos. Con suerte esto debería funcionar.

Notar que estos son métodos Python normales. Agreguemos uno más, como demostración:

```
import datetime
from django.utils import timezone
# ...
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Notar que agregamos `import datetime` y `from django.utils import timezone`, para referenciar el módulo `datetime` de la librería estándar de Python y las utilidades de Django relacionadas a zonas horarias en `django.utils.timezone`, respectivamente. Si no estás familiarizado con el manejo de zonas horarias en Python, podés aprender más en la `documentación de time zone`.

Guardamos los cambios y empezamos una nueva sesión en el shell corriendo `python manage.py shell` nuevamente:

```
>>> from polls.models import Poll, Choice

# Confirmemos que __unicode__() funciona.
>>> Poll.objects.all()
[<Poll: What's up?>]

# Django provee una API de búsqueda muy completa a partir de keywords arguments.
>>> Poll.objects.filter(id=1)
[<Poll: What's up?>]
>>> Poll.objects.filter(question__startswith='What')
[<Poll: What's up?>]

# Obtengamos la encuesta que se publicó este año.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Poll.objects.get(pub_date__year=current_year)
<Poll: What's up?>

# Si pedimos por un ID que no existe, se levantará una excepción.
>>> Poll.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Poll matching query does not exist. Lookup parameters were {'id': 2}

# Buscar por clave primaria es un caso muy común, Django provee un atajo
# para búsquedas exactas por primary-key.
# Lo siguiente es equivalente a Poll.objects.get(id=1).
>>> Poll.objects.get(pk=1)
<Poll: What's up?>

# Confirmemos que el método que agregamos funciona.
>>> p = Poll.objects.get(pk=1)
>>> p.was_published_recently()
True
```

```
# Démosle a la Poll un par de Choices. La llamada a create crea un nuevo
# objeto Choice, ejecuta la sentencia INSERT, agrega la opción al conjunto
# de opciones disponibles y devuelve el nuevo objeto Choice. Django crea
# un conjunto para mantener el "otro lado" de la relación dada por ForeignKey
# (e.g., las opciones de la encuesta) que se puede acceder vía la API.
>>> p = Poll.objects.get(pk=1)

# Muestra las opciones relacionadas -- ninguna por ahora.
>>> p.choice_set.all()
[]

# Creamos tres opciones.
>>> p.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice_text='Just hacking again', votes=0)

# Los objetos Choice tienen acceso vía la API a la Poll relacionada.
>>> c.poll
<Poll: What's up?>

# Y vice versa: los objetos Poll tienen acceso a los objetos Choice.
>>> p.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> p.choice_set.count()
3

# La API sigue las relaciones automáticamente tanto como se necesite.
# Se usa doble guión bajo para separar relaciones.
# Esto funciona tantos niveles de profundidad como se quiera; no hay límite.
# Encontremos todas las Choices para cualquier Poll cuya pub_date es
# este año (reusando la variable 'current_year' que creamos arriba).
>>> Choice.objects.filter(poll__pub_date__year=current_year)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# Borremos una de las opciones. Para esto usamos delete().
>>> c = p.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
```

Para más información sobre relaciones en modelos, ver `Acceder objetos relacionados`. Para más detalles sobre cómo usar los doble guión bajo para efectuar búsquedas usando la API, ver *Field lookups*. Para el detalle completo de la API de base de datos, ver `Database API reference`.

Cuando te sientas confortable con la API, podés pasar a *parte 2 del tutorial* para tener funcionando la interfaz automática de administración de Django.

1.4 Escribiendo tu primera Django app, parte 2

Este tutorial comienza donde dejó *Tutorial 1*. Continuaremos con la aplicación de encuestas y nos concentraremos en la interfaz de administración (que llamaremos 'admin') que Django genera automáticamente.

Filosofía

Generar sitios de administración para que gente de staff o clientes agreguen, cambien y borren contenido es un trabajo tedioso que no requiere mucha creatividad. Por esta razón, Django automatiza completamente la creación de una

interfaz de administración para los modelos.

Django fue escrito en el ambiente de una sala de noticias, con una separación muy clara entre "administradores de contenido" y el sitio "público". Los administradores del sitio usan el sistema para agregar noticias, eventos, resultados deportivos, etc., y el contenido se muestra en el sitio público. Django resuelve el problema de crear una interfaz unificada para que los administradores editen contenido.

El admin no está pensado para ser usado por los visitantes de un sitio, sino para los administradores del mismo.

1.4.1 Activando el admin

El admin de Django no está habilitado por defecto – es opcional. Para activarlo hay que seguir estos tres pasos:

- Descomentar `"django.contrib.admin"` en el setting `INSTALLED_APPS`.
- Correr `python manage.py syncdb`. Como agregamos una nueva aplicación a `INSTALLED_APPS`, necesitamos actualizar las tablas de la base de datos.
- Editar el archivo `mysite/urls.py` y descomentar las líneas que hacen referencia al admin – hay tres líneas que descomentar en total. Este archivo es un URLconf; profundizaremos en URLconfs en el siguiente tutorial. Por ahora, todo lo que necesitamos saber es que mapea URLs raíces a aplicaciones. Al terminar deberíamos tener un archivo `urls.py` que se vería algo así:

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', '{{ project_name }}.views.home', name='home'),
    # url(r'^{{ project_name }}/', include('{{ project_name }}.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

(Las líneas en negrita son las que hace falta descomentar)

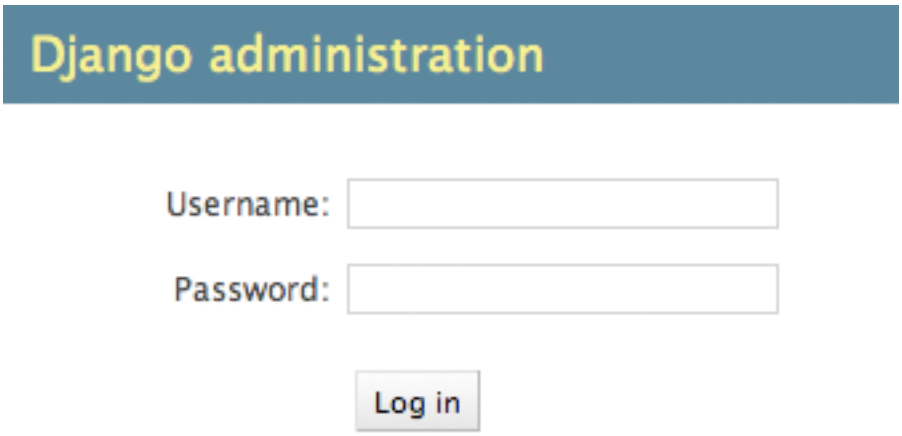
1.4.2 Levantar el servidor de desarrollo

Levantemos el servidor de desarrollo y exploremos la interfaz del admin.

Recordemos del Tutorial 1 que para empezar el servidor de desarrollo debemos correr:

```
python manage.py runserver
```

Ahora, abrimos un browser y vamos a "/admin/" en el dominio local – e.g., http://127.0.0.1:8000/admin/. Deberíamos ver la pantalla de login del admin:



No es lo que estás viendo?

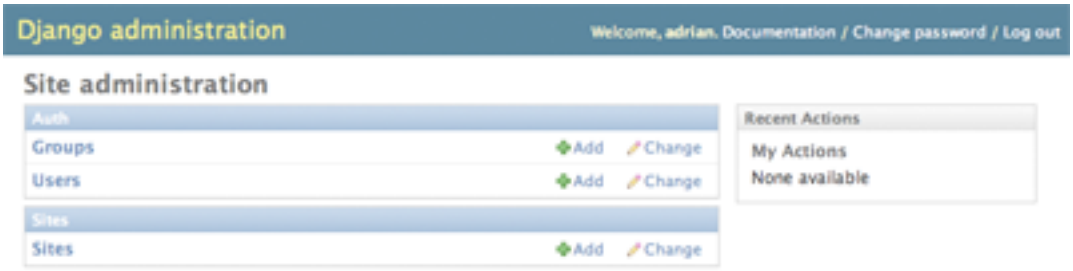
Si en este punto, en lugar de la página de login de arriba, obtenés una página de error parecida a:

```
ImportError at /admin/
cannot import name patterns
...
```

es probable que estés usando una versión de Django que no coincide con la de este tutorial. Deberías o bien cambiar a una versión anterior del tutorial, o bien actualizar la versión de Django.

1.4.3 Ingresar al admin

Ahora intentemos loguearnos (creamos una cuenta de superusuario en la primera parte de este tutorial, te acordás? Si no lo hiciste, o te olvidaste el password podés *crear otro*). Deberíamos ver la página inicial del admin de Django:



Deberíamos ver unos pocos tipos de contenido editable, incluyendo grupos, usuarios y sitios. Éstos vienen con Django por defecto.

1.4.4 Hacer la app poll modificable en el admin

Dónde está nuestra app poll? No se muestra en la página del admin.

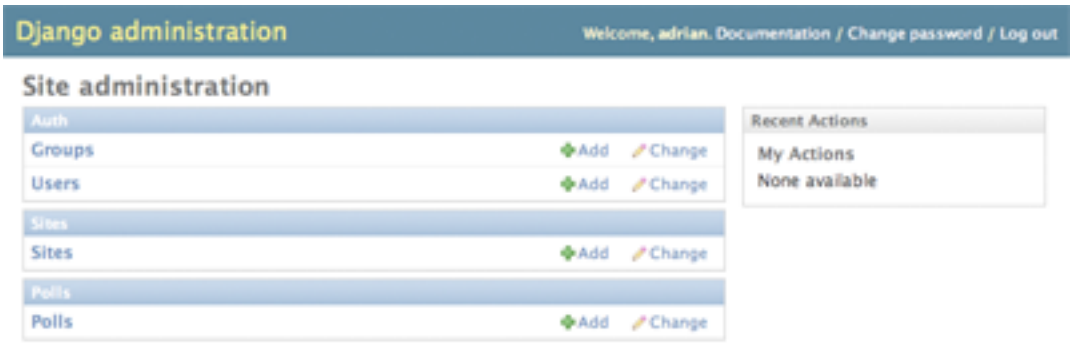
Hay una cosa que hacer: necesitamos decirle al admin que los objetos `Poll` tengan una interfaz de administración. Para esto creamos un archivo llamado `admin.py` en el directorio `polls`, y lo editamos para que tenga el siguiente contenido:

```
from django.contrib import admin
from polls.models import Poll

admin.site.register(Poll)
```

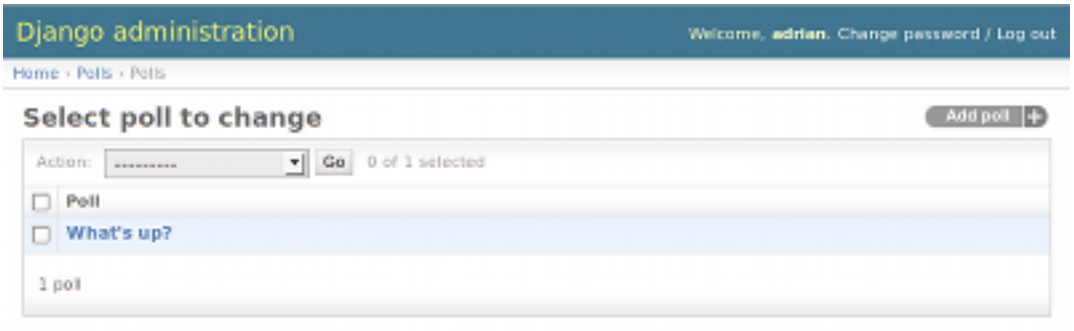
Necesitaremos reiniciar el servidor de desarrollo para ver los cambios. Normalmente el servidor de desarrollo se reinicia cada vez que modificamos un archivo, pero no con la creación de uno nuevo.

1.4.5 Explorar la funcionalidad del admin

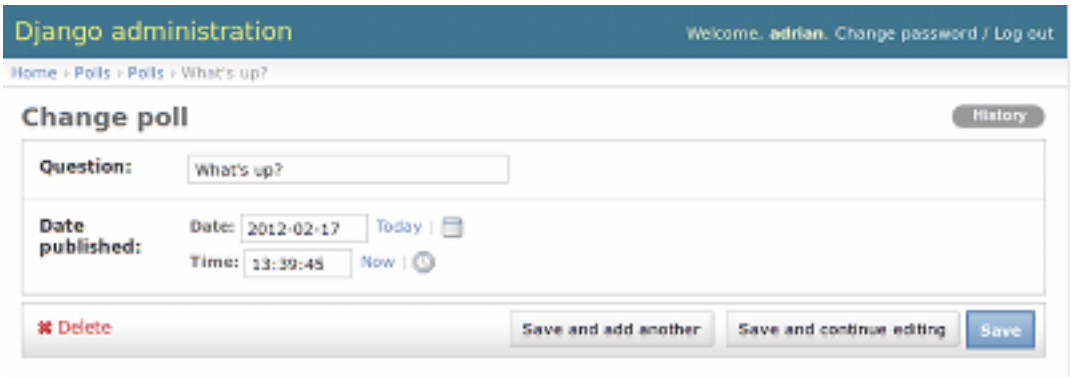
Ahora que registramos `Poll`, Django sabe que se debe mostrar en la página del admin:



Hacemos click en "Polls". Ahora estamos en la página de listado (change list) de encuestas. Esta página muestra todas las encuestas en la base de datos y nos permite elegir una para modificarla. Está la encuesta "What's up" que creamos en la primera parte:



Cliqueamos en ella para editarla:



Cosas para notar aquí:

- El form es generado automáticamente a partir del modelo Poll.
- Los diferentes tipos de campo del modelo (`DateTimeField`, `CharField`) se corresponden con el widget HTML apropiado. Cada tipo de campo sabe cómo mostrarse en el admin de Django.
- Cada `DateTimeField` tiene atajos JavaScript. La fecha tienen un atajo para "Hoy" (Today) y un popup de calendario, y la hora un "Ahora" (Now) y un popup que lista las horas más comunes.

El cierre de la página nos da algunas opciones:

- Save – Guarda los cambios y nos devuelve a la página listado para este tipo de objeto.
- Save and continue editing – Guarda los cambios y recarga la página del admin de este objeto.
- Save and add another – Guarda los cambios y carga un form nuevo, en blanco, que permite agregar un nuevo objeto de este tipo.
- Delete – Muestra una página de confirmación de borrado.

Si el valor de "Date published" no coincide con el establecido durante la creación del objeto en Tutorial 1, probablemente sea que olvidaste setear el valor correcto para el `TIME_ZONE`. Revisalo, recargá la página y chequeá que se muestra el valor correcto.

Cambiamos el valor de "Date published" haciendo click en "Today" y "Now". Luego hacemos click en "Save and continue editing". Si hacemos click en "History" arriba a la derecha, podemos ver una página que lista todos los cambios hechos a este objeto a través del admin, con la fecha/hora y el nombre de usuario de la persona que hizo el cambio:

1.4.6 Personalizar el form del admin

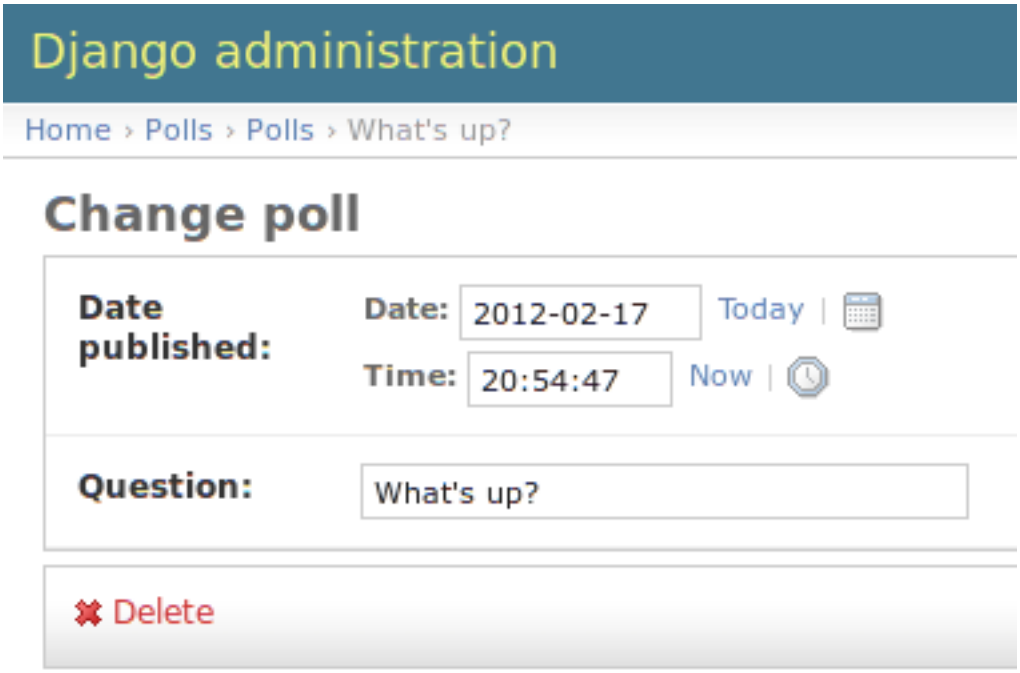
Lleva unos pocos minutos maravillarse por todo el código que no tuvimos que escribir. Sólo registrando el modelo Poll con `admin.site.register(Poll)`, Django fue capaz de construir una representación con un form por defecto. A menudo uno querrá personalizar cómo este form se ve y funciona. Esto lo hacemos pasando algunas opciones a Django cuando registramos el modelo.

Veamos cómo funciona reordenando los campos en el form de edición. Reemplazamos la línea `admin.site.register(Poll)` por:

```
class PollAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question']

admin.site.register(Poll, PollAdmin)
```

Seguiremos este patrón – creamos una clase ModelAdmin, que luego pasamos como segundo argumento de `admin.site.register()` – cada vez que necesitemos cambiar alguna opción del admin para un modelo.

Este cambio en particular hace que "Publication date" se muestre antes que el campo "Question":

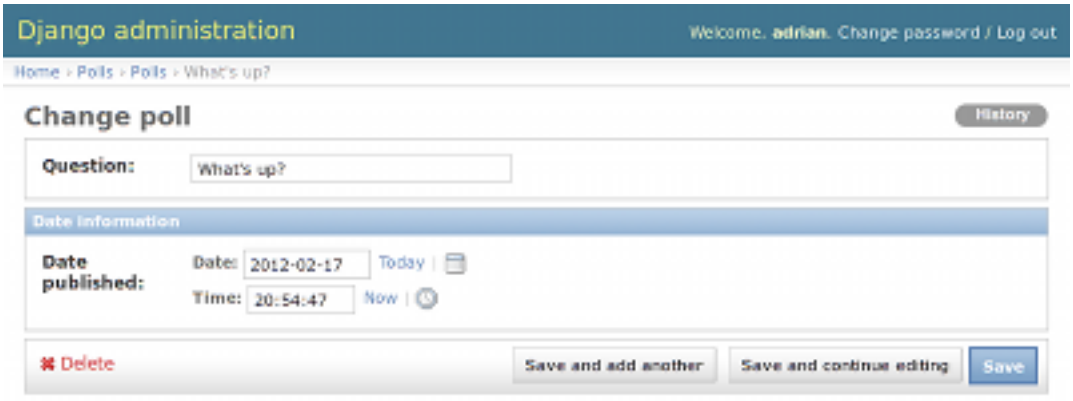


No es muy impresionante con sólo dos campos, pero para forms con docenas de campos, elegir un orden intuitivo es un detalle de usabilidad importante.

Y hablando de forms con docenas de campos, podríamos querer dividir el form en fieldsets:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Poll, PollAdmin)
```

El primer argumento de cada tupla en `fieldsets` es el título del fieldset. Ahora nuestro form se vería así:



Podemos asignar clases HTML arbitrarias a cada fieldset. Django provee una clase `collapse"` que muestra el fieldset inicialmente plegado. Esto es útil cuando uno tiene un form largo que contiene ciertos campos que no se usan normalmente:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
```

1.4.7 Agregando objetos relacionados

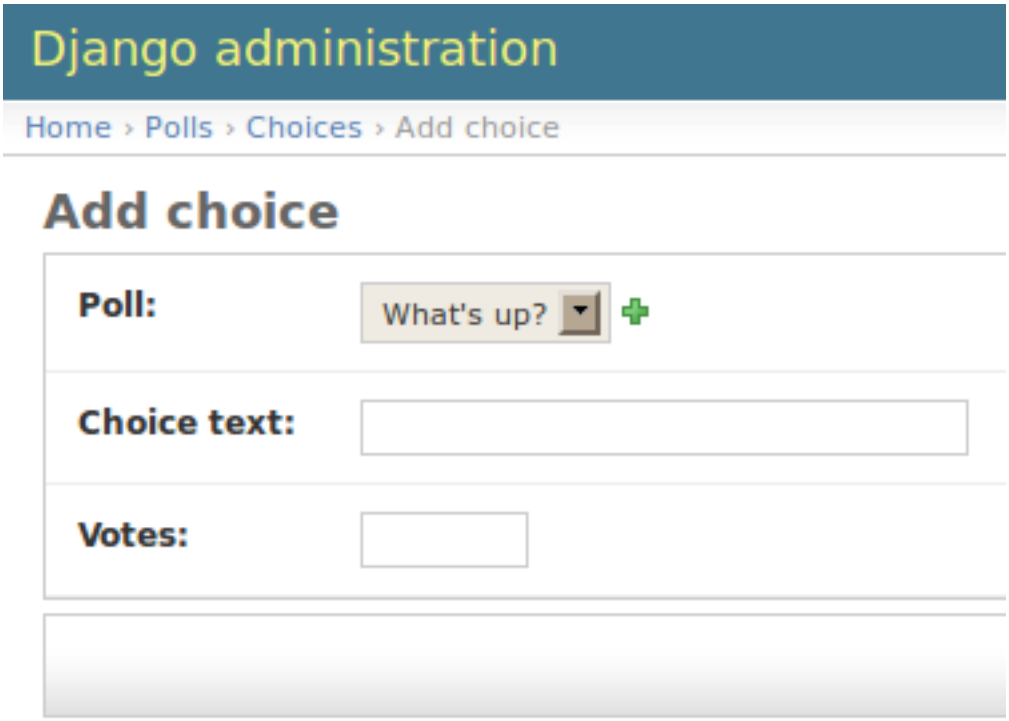
OK, tenemos nuestra página de admin para el modelo Poll. Pero un objeto `Poll` tiene múltiple `Choices`, y la página de admin no nos muestra estas opciones.

Todavía.

Hay dos formas de resolver este problema. La primera es registrar `Choice` con el admin como hicimos con `Poll`. Esto es fácil:

```
from polls.models import Choice

admin.site.register(Choice)
```

Ahora "Choices" está disponible en el admin de Django. El form de "Add choice" se vería como esto:



En ese form, el campo "Poll" es un desplegable que contiene todas las encuestas en la base de datos. Django sabe que un `ForeignKey` debe mostrarse como un `<select>` en el admin. En nuestro caso, existe solamente una encuesta en este momento.

Notemos también el link "Add Another" al lado de "Poll". Todo objeto con una relación de `ForeignKey` a otro tiene esta opción, gratis. Cuando uno hace click en "Add Another", se obtiene un popup con el form de "Add poll". Si uno agrega una encuesta mediante este form y hace click en "Save", Django va a guardar la encuesta en la base de datos y dinámicamente seleccionarla como valor en el form de "Add choice" que estábamos viendo.

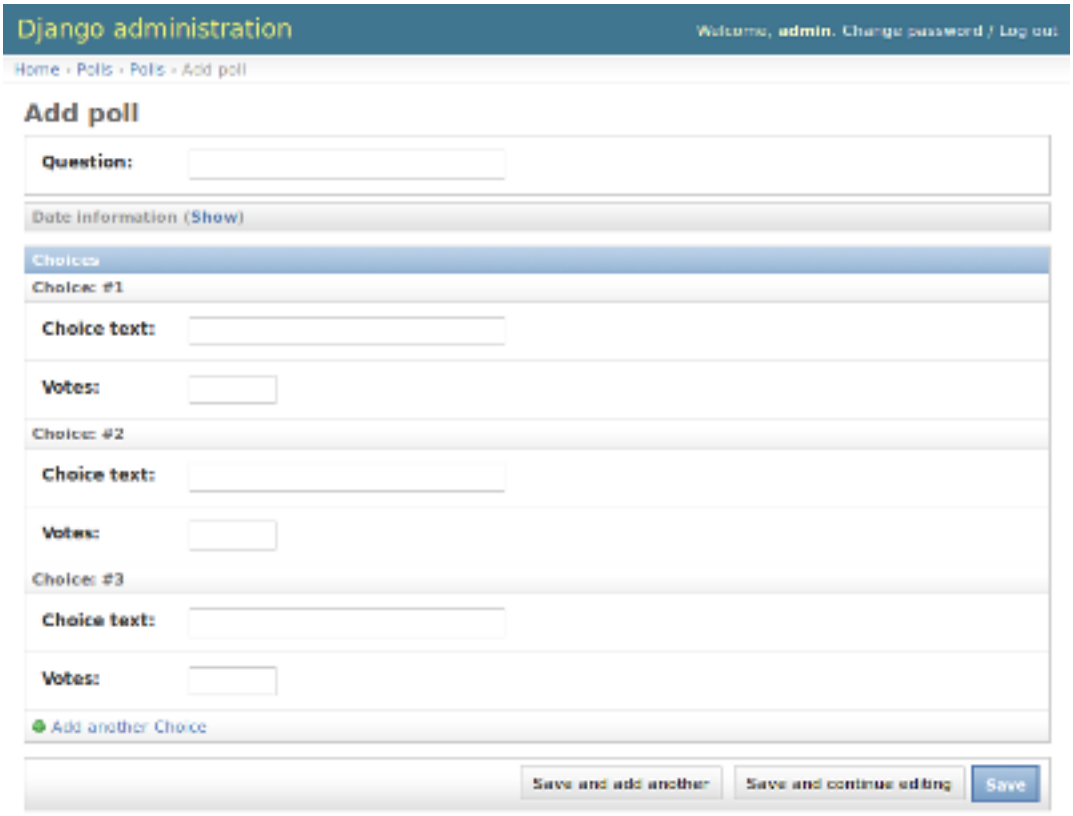
Pero, la verdad, esta es una manera ineficiente de agregar objetos `Choice` al sistema. Sería mejor su uno pudiera agregar varias Choices directamente cuando se crea un objeto `Poll`. Hagamos que eso ocurra.

Borramos la llamada a `register()` para el modelo `Choice`. Editamos el código que registra `Poll` para que se vea así:

```
from django.contrib import admin
from polls.models import Choice, Poll

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

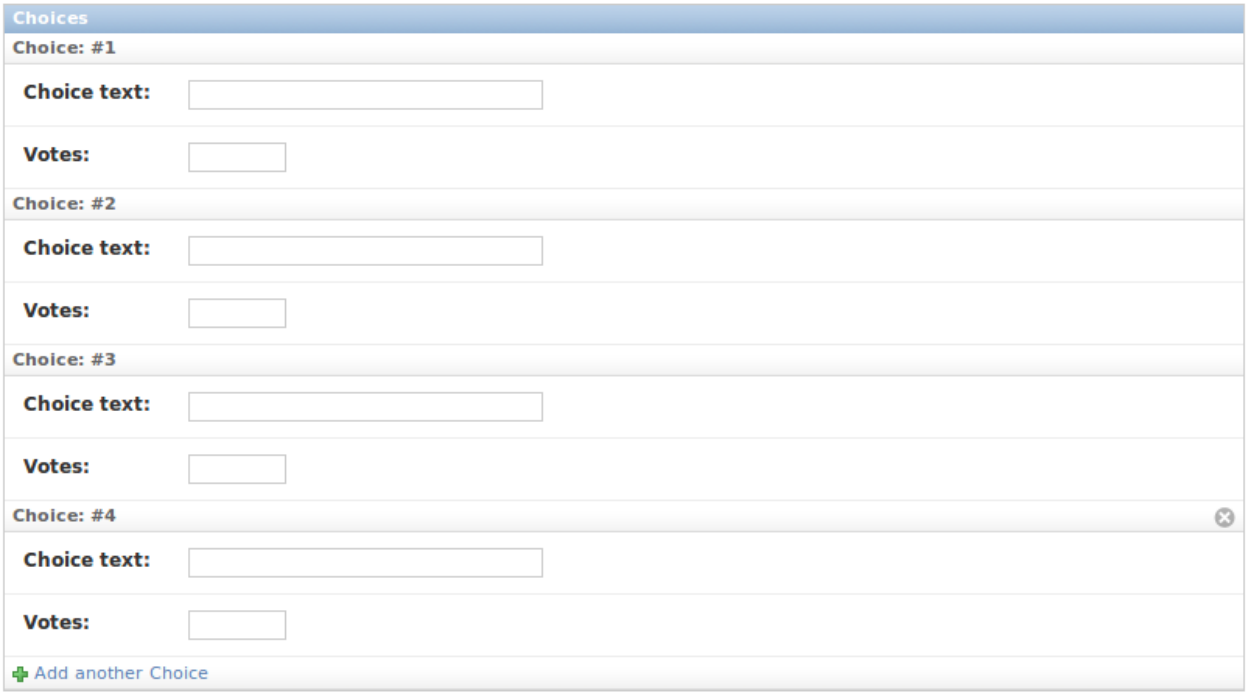
Esto le dice a Django: "los objetos `Choice` se editan en la página de admin de `Poll`. Por defecto, proveer los campos para 3 choices".

Cargamos la página de "Add poll" para ver cómo se ve, puede ser necesario reiniciar el servidor de desarrollo:



It works like this: There are three slots for related Choices – as specified by `extra` – and each time you come back to the "Change" page for an already-created object, you get another three extra slots.

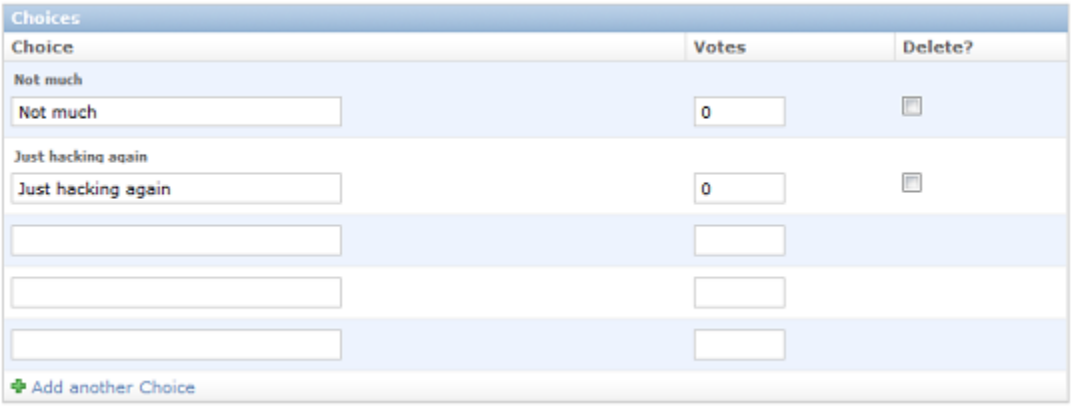
At the end of the three current slots you will find an "Add another Choice" link. If you click on it, a new slot will be added. If you want to remove the added slot, you can click on the X to the top right of the added slot. Note that you can't remove the original three slots. This image shows an added slot:



Un pequeño problema. Ocupa mucho espacio en pantalla mostrar todos los campos para ingresar los objetos `Choice` relacionados. Por esa razón, Django ofrece una forma tabular para mostrar objetos relacionados "inline"; solamente necesitamos cambiar la declaración de `ChoiceInline` de esta manera:

```
class ChoiceInline(admin.TabularInline):
    #...
```

Con `TabularInline` (en lugar de `StackedInline`), los objetos relacionados se muestran de forma más compacta, en forma de tabla:

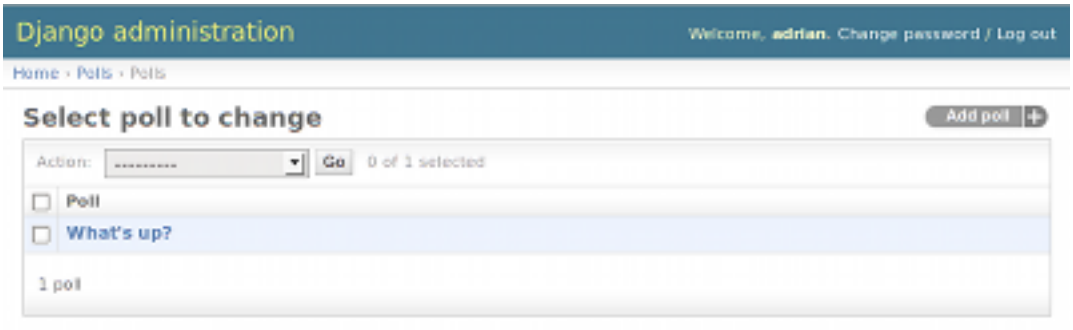


Notemos que hay una columna extra con la opción "Delete?" que nos permite borrar filas que hayamos agregado usando el botón "Add Another Choice" y filas que ya hayan sido guardadas.

1.4.8 Personalizar la página listado

Ahora que la página de admin de `Poll` se ve bien, hagamos algunos tweaks a la página de listado – la que lista todas las encuestas en el sistema.

Así se ve ahora:



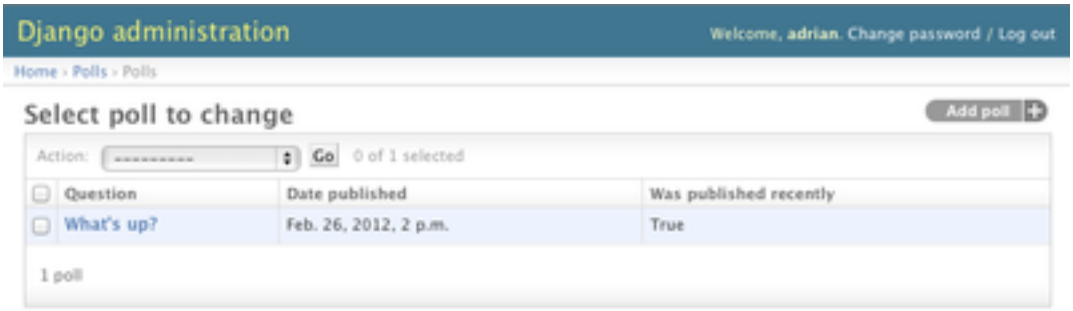
Por defecto, Django muestra el `str()` de cada objeto. Pero algunas veces es más útil si podemos mostrar campos individuales. Para eso usamos la opción `list_display` del admin, que es una tupla de los nombres de campo a mostrar, como columnas, en la página de listado:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date')
```

Sólo por si acaso, incluyamos también el método `was_published_recently` que definimos en la primera parte:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date', 'was_published_recently')
```

Ahora la página de listado de encuestas se vería así:



Podemos hacer click en los encabezados de las columnas para ordenar por los respectivos valores – salvo en el caso de `was_published_recently`, porque ordenar por la salida de un método arbitrario no está soportado. Notemos también que el encabezado para la columna de `was_published_recently` es por defecto el nombre del método (reemplazando guiones bajos por espacios), y que cada línea contiene la representación como string del valor devuelto por el método.

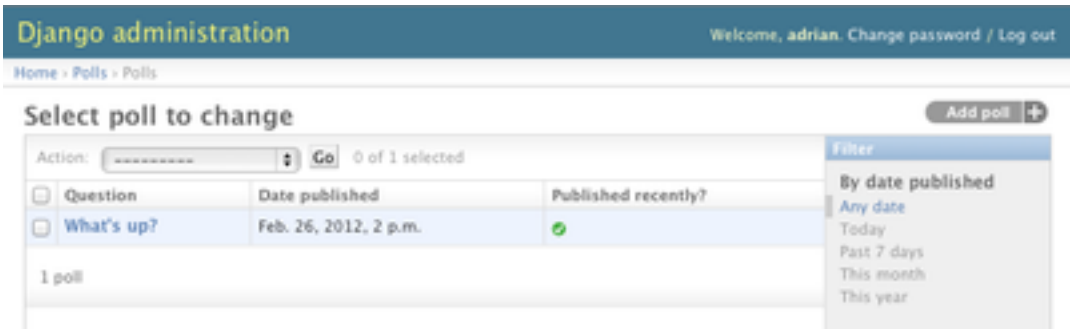
Esto se puede mejorar definiendo un par de atributos del método (en `models.py`) como sigue:

```
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

Editamos el archivo `admin.py` de nuevo y agregamos una mejora a la página de listado de encuestas: Filtros. Agregamos la siguiente línea a `PollAdmin`:

```
list_filter = ['pub_date']
```

Esto agrega un "Filtro" en la barra lateral que permite filtrar el listado por el campo `pub_date`:



El tipo de filtro depende del tipo de campo sobre el cual se filtra. Como `pub_date` es un `DateTimeField`, Django sabe darnos opciones de filtro apropiadas: "Any date", "Today", "Past 7 days", "This month", "This year".

Esto está tomando buena forma. Agreguemos campacidad de búsqueda:

```
search_fields = ['question']
```

Esto agrega un campo de búsqueda al tope del listado. Cuando alguien ingresa términos de búsqueda, Django va a buscar sobre el campo `question`. Se pueden usar tantos campos como uno quiera – aunque como por detrás se trata de una consulta del tipo `LIKE` hay que ser razonables para mantener feliz a la base de datos.

Finalmente, como los objetos `Poll` tienen fechas, sería conveniente poder navegarlos por fecha. Agregamos esta línea:

```
date_hierarchy = 'pub_date'
```

Esto agrega una barra de navegación jerárquica por fecha, al tope del listado. En el primer nivel lista todos los años disponibles. Luego permite filtrar por mes y finalmente por días.

Es un buen momento para observar también que el listado es paginado. Por defecto se muestran 100 items por página. Paginación, búsqueda, filtros, jerarquía de fechas y ordenamiento de columnas por encabezado funcionan como uno esperaría.

1.4.9 Personalizar el "look and feel" del admin

Claramente tener el título "Django administration" al tope de cada página del admin es ridículo. Es un "placeholder".

Es fácil de cambiar usando el sistema de templates de Django. El admin de Django funciona gracias a Django mismo, y la interfaz provista usa el sistema de templates de Django.

Abrimos el archivo de settings (recordemos, `mysite/settings.py`) y vemos el setting `TEMPLATE_DIRS`. Se trata de una tupla de directorios del sistema de archivos para chequear cuando Django carga los templates. Es un path de búsqueda.

Creamos un directorio `mytemplates` en el directorio del proyecto. Los templates pueden estar en cualquier lugar de nuestro sistema de archivos al que Django tenga acceso (Django corre como el usuario que corra el servidor). Sin embargo, mantener los templates dentro del proyecto es una buena convención a seguir.

Por defecto, `TEMPLATE_DIRS` está vacío. Entonces agreguemos una nueva línea para decirle a Django dónde encontrar nuestros templates:

```
TEMPLATE_DIRS = (
    '/path/to/mysite/mytemplates', # Cambiar esto al directorio que corresponda.
)
```

Ahora copiamos el template `admin/base_site.html` del directorio de templates del admin de Django (`django/contrib/admin/templates`) en un subdirectorio `admin` del directorio que hayamos agregado a `TEMPLATE_DIRS`. Por ejemplo, si nuestro `TEMPLATE_DIRS` incluye `'/path/to/mysite/mytemplates'`, como arriba, copiamos `django/contrib/admin/templates/admin/base_site.html` a `/path/to/mysite/mytemplates/admin/base_site.html`. No olvidarse el subdirectorio `admin`.

Dónde están los archivos del código fuente de Django?

Si tenés problemas encontrando dónde están los archivos de Django, podés correr el siguiente comando:

```
python -c "
import sys
sys.path = sys.path[1:]
import django
print(django.__path__)"
```

Luego, basta editar el archivo y reemplazar el texto genérico sobre Django por el nombre de nuestro sitio.

Este archivo de template contiene varios textos de la forma `{% block branding %}` y and `{{ title }}`. Los tags `{%` y `{{` son parte del lenguaje de templates de Django. Cuando Django renderiza `admin/base_site.html`, este lenguaje se evalúa para producir la página HTML final. No nos preocupemos de esto por ahora – veremos en detalle el lenguaje de templates de Django en Tutorial 3.

Notemos que cualquier template del admin de Django se puede "sobreescribir". Para esto sólo basta repetir lo que hicimos con `base_site.html` – copiar el template desde el directorio de Django a nuestro directorio de templates (respetando el path relativo, dentro de `admin`) y aplicar los cambios.

Los lectores astutos se preguntarán: pero si `TEMPLATE_DIRS` estaba vacío por defecto, cómo encuentra Django los templates del admin? La respuesta es que, por defecto, Django chequea automáticamente por un subdirectorio `templates/` dentro de cada app, para usar como fallback. Para mayor información se puede consultar la *documentación del template loader*.

1.4.10 Personalizar la página inicial del admin

De forma similar uno podría querer personalizar el look and feel de la página inicial del admin.

Por defecto, muestra todas las apps en `INSTALLED_APPS` que se registraron con la aplicación admin, en orden alfabético. Uno podría querer hacer cambios significativos al layout. Después de todo, la inicial es probablemente la página más importante del admin, y debería ser fácil de usar.

El template a personalizar es `admin/index.html` (habría que hacer lo mismo que hicimos en la sección anterior con `admin/base_site.html` – copiar el template a nuestro directorio de templates). Editamos el archivo, y veremos que usa una variable de template llamada `app_list`. Esta variable contiene cada app instalada. En lugar de usarla, uno podría escribir los links de manera explícita a las páginas de objetos específicas del admin de la manera en que a uno le parezca mejor. De nuevo, no preocuparse si no se entiende el lenguaje de templates – lo cubriremos en detalle en Tutorial 3.

Una vez que te sientas cómodo con el sitio de admin, podés empezar con la *parte 3 del tutorial* donde comenzamos a trabajar con las vistas públicas de encuestas.

1.5 Escribiendo tu primera Django app, parte 3

Este tutorial empieza donde quedó el *Tutorial 2*. Continuamos con la aplicación web para encuestas y nos concentraremos en la creación de la interfaz pública – "views".

1.5.1 Filosofía

Una view es un "tipo" de página web en nuestra aplicación Django que generalmente provee una función específica y tiene un template particular. Por ejemplo, en una aplicación de blog, uno podría tener las siguientes views:

- Blog homepage – muestra las últimas entradas.
- Detalle de una entrada – página correspondiente a una entrada o post.
- Archivo anual – muestra los meses de una año con sus correspondientes entradas.
- Archivo mensual – muestra los días de un mes con las correspondientes entradas.
- Archivo diario – muestra todas las entradas en un día dado.
- Acción de comentar – maneja el posteo de comentarios en una entrada dada.

En nuestra aplicación de encuestas, vamos a tener las siguientes cuatro views:

- Página inicial – muestra las últimas encuestas.
- Detalle de encuesta – muestra la pregunta de la encuesta, sin resultados, junto con un form para votar.
- Página de resultados – muestra los resultados para una encuesta particular.
- Acción de votar – maneja el voto por una opción particular en una encuesta dada.

En Django, cada view se representa mediante una simple función Python.

1.5.2 Escribiendo nuestra primera view

Escribamos nuestra primera view. Abrimos el archivo `polls/views.py` y ponemos el siguiente código Python:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the poll index.")
```

Esta es la view más simple posible en Django. Pero tenemos un problema, cómo se llama a esta view? Para eso necesitamos mapearla a una URL, en Django lo hacemos mediante un archivo de configuración llamado URLconf.

Qué es un URLconf?

En Django, las páginas web y cualquier otro contenido se entrega a partir de views, y quien determinar qué view se llama es un módulo Python informalmente llamado 'URLconfs'. Estos módulos son código Python puro y son un simple mapeo de patrones de URL (expresados como expresiones regulares) a funciones Python de callback (las views). Este tutorial provee las instrucciones básicas para su uso, para mayor información se puede recurrir a `django.core.urlresolvers`.

Para crear un URLconf, en el directorio polls creamos un archivo llamado `urls.py`. La estructura de directorios de nuestra app ahora debería verse así:

```
polls/
    __init__.py
    admin.py
    models.py
    tests.py
    urls.py
    views.py
```

En el archivo `polls/urls.py` incluimos el siguiente código:

```
from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',
    url(r'^$', views.index, name='index')
)
```

El próximo paso es apuntar el URLconf raíz al módulo `polls.urls`. En `mysite/urls.py` insertamos un `include()`, con lo que nos quedaría:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

Hemos conectado una view *index* en el URLconf. Si vamos a la dirección http://localhost:8000/polls/ en nuestro browser, deberíamos ver el texto "*Hello, world. You're at the poll index.*", que definimos la view `index`.

La función `url()` toma cuatro argumentos, dos requeridos: `regex` y `view`, y dos opcionales: `kwargs`, y `name`. En este punto vale la pena revisar para que es cada uno de ellos.

`url()`: argumento regex

El término *regex* se usa comúnmente como abreviatura para *expresión regular* (*regular expression*, en inglés), que es una forma de describir patrones que se verifiquen en un string, o en este caso patrones de url. Django comienza en la primera expresión regular y va recorriendo la lista hacia abajo, comparando la URL solicitada contra cada expresión regular hasta encontrar una cuyo patrón coincida.

Notar que estas expresiones regulares no chequean parámetros de GET o POST, o el nombre de dominio. Por ejemplo, en un pedido por `http://www.example.com/myapp/`, el URLconf buscará por `myapp/`. En un pedido por `http://www.example.com/myapp/?page=3`, el URLconf también buscará por `myapp/`.

Si necesitaras ayuda con las expresiones regulares, podés ver Wikipedia y la documentación del módulo `re`. También es fantástico, el libro de O'Reilly "Mastering Regular Expressions", de Jeffrey Friedl. En la práctica, sin embargo, no hace falta ser un experto en expresiones regulares, ya que lo que realmente es necesario saber es cómo capturar patrones simples. De hecho, expresiones regulares complejas pueden afectar la performance de búsqueda, por lo que uno no debería confiar en todo el poder de las expresiones regulares.

Finalmente, una nota sobre performance: estas expresiones regulares se compilan la primera vez que el módulo se carga. Son súper rápidas (mientras que no sean complejas como se menciona arriba).

`url()`: argumento view

Cuando Django encuentra una expresión regular que coincide, se llama a la función view especificada, con un objeto `HttpRequest` como primer argumento y los valores que se hayan "capturado" a partir de la expresión regular como argumentos restantes. Si la regex usa capturas simples (sin nombre), los valores se pasan como argumentos posicionales; si se usan capturas con nombre, los valores se pasan como argumentos nombrados. Veremos un ejemplo en breve.

`url()`: argumento kwargs

Se pueden pasar argumentos arbitrarios en un diccionario a la view de destino. No vamos a usar esta opción en el tutorial.

`url()`: argumento name

Nombrar las URL nos permite referirnos a ellas de forma unívoca desde distintas partes del código, especialmente en los templates. Esta característica nos permite hacer cambios globales a los patrones de url del proyecto cambiando tan sólo un archivo (ya que el nombre permanece sin cambios, y nos referimos a una URL po su nombre).

1.5.3 Escribiendo más views

Ahora agreguemos algunas views más a `polls/views.py`. Estas views van a ser un poco diferentes porque van a tomar un argumento:

```
def detail(request, poll_id):
    return HttpResponse("You're looking at poll %s." % poll_id)

def results(request, poll_id):
    return HttpResponse("You're looking at the results of poll %s." % poll_id)

def vote(request, poll_id):
    return HttpResponse("You're voting on poll %s." % poll_id)
```

Conectemos estas nuevas views en el módulo `polls.urls` agregando las siguientes llamadas a `url()`:

```
from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<poll_id>\d+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<poll_id>\d+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<poll_id>\d+)/vote/$', views.vote, name='vote'),
)
```

Veamos en nuestro browser "/polls/34/". Se ejecutará la función `detail()` y nos mostrará el ID que pasamos en la URL. Probemos también "/polls/34/results/" y "/polls/34/vote/" – esto nos debería mostrar los placeholder que definimos para las páginas de resultados y para votar.

Cuando alguien pide por una página de nuestro sitio – supongamos "/polls/34/", Django va a cargar el módulo `mysite.urls`, al que apunta el setting `ROOT_URLCONF`. Encuentra la variable `urlpatterns` y recorre las expresiones regulares en orden. Las llamadas a `include()` simplemente referencian otros URLconf. Notar que las expresiones regulares para los `include()` no tienen un `$` (caracter que indica el fin de string en un patrón), sino que terminan en una barra. Cada vez que Django encuentra un `include()`, recorta la parte de la URL que coincide hasta ese punto y envía el string restante al URLconf relacionado para continuar el proceso.

La idea detrás de `include()` es hacer fácil tener URLs plug-and-play. Como polls tiene su propio URLconf (`polls/urls.py`), las URLs de la app se pueden poner bajo "/polls/", o bajo "/fun_polls/", o bajo "/content/polls/", o cualquier otro camino, y la app seguirá funcionando.

Esto es lo que pasa si un usuario va a "/polls/34/" en este sistema:

- Django encontrará coincidencia en `'^polls/'`
- Entonces, Django va a recortar el texto que coincide (`"polls/"`) y enviar el texto restante – `"34/"` – al URLconf 'polls.urls' para seguir el proceso, donde coincidirá con `r'^(?P<poll_id>\d+)/$'`, resultando en una llamada a la view `detail()` de la forma:

  ```
  detail(request=<HttpRequest object>, poll_id='34')
  ```

La parte `poll_id='34'` surge de `(?P<poll_id>\d+)`. Usando paréntesis alrededor de un patrón se "captura" el texto que coincida con el patrón y ese valor se pasas como argumento a la función view; `?P<poll_id>` define el nombre que se usará para identificar la coincidencia; y `\d+` es una expresión regular para buscar una secuencia de dígitos (i.e., un número).

Como los patrones de URL son expresiones regulares, no hay realmente un límite de lo que se puede hacer con ellos. Y no hay necesidad de agregar cosas como `.html` – a menos que uno quisiera, en cuyo caso nos quedaría algo como:

```
(r'^polls/latest\.html$', 'polls.views.index'),
```

Pero no hagan esto. No tiene sentido.

1.5.4 Escribiendo views que hacen algo

Cada view es responsable de hacer una de dos cosas: devolver un objeto `HttpResponse` con el contenido de la página solicitada, o levantar una excepción, por ejemplo `Http404`. El resto depende de uno.

Una view puede leer registros de una base de datos, o no. Puede usar un sistema de templates como el de Django – o algún otro basado en Python –, o no. Puede generar un archivo PDF, una salida XML, crear un archivo ZIP, cualquier cosa que uno quiera, usando cualquier librería Python que uno quiera.

Todo lo que Django espera es un `HttpResponse`. O una excepción.

Por ser conveniente, vamos a usar la API de Django para base de datos, que vimos en el *Tutorial 1*. Aquí tenemos una aproximación a la view `index()` que muestra las 5 encuestas más recientes en el sistema, separadas por comas, de acuerdo a la fecha de publicación:

```
from django.http import HttpResponse

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.order_by('-pub_date')[:5]
    output = ', '.join([p.question for p in latest_poll_list])
    return HttpResponse(output)
```

Pero tenemos un problema: el diseño de la página está escrito explícitamente en la view. Si uno quisiera cambiar cómo se ve la página, debería editar el código Python. Entonces vamos a usar el sistema de templates de Django para separar el diseño del código Python.

Primero, creamos un directorio `polls` en el directorio de templates especificado en setting:*TEMPLATE_DIRS*. Allí creamos un archivo llamado `index.html`. En ese template escribimos el siguiente código:

```
{% if latest_poll_list %}
    <ul>
    {% for poll in latest_poll_list %}
        <li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
    {% endfor %}
    </ul>
{% else %}
```

```
    <p>No polls are available.</p>
{% endif %}
```

Ahora usemos ese template html en nuestra view `index`:

```
from django.http import HttpResponse
from django.template import Context, loader

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = Context({
        'latest_poll_list': latest_poll_list,
    })
    return HttpResponse(template.render(context))
```

Este código carga el template llamado `polls/index.html` y le pasa un contexto. El contexto es un diccionario que mapea nombres de variable a nivel template a objetos Python.

Si cargamos la página en nuestro browser, deberíamos ver una lista conteniendo la encuesta "What's up" del Tutorial 1. El link nos lleva a la página de detalle de la encuesta.

Organizando los templates

En lugar de tener un gran directorio de templates, uno puede guardar templates dentro de cada app. Veremos esto en más detalle en el *tutorial sobre apps reusables*.

Un atajo: `render()`

La acción de cargar un template, armar un contexto y devolver un objeto `HttpResponse` con el resultado de renderizar el template es muy común. Django provee un atajo. Aquí está la view `index()` reescrita:

```
from django.shortcuts import render

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    context = {'latest_poll_list': latest_poll_list}
    return render(request, 'polls/index.html', context)
```

Una vez que hayamos hecho esto para todas estas views, ya no necesitamos importar `loader`, `Context` y `HttpResponse` (habrá que mantener `HttpResponse` si es que todavía tenemos métodos stub para `detail`, `results` y `vote`).

La función `render()` toma un objeto `request` como primer un argumento, un nombre de template como segundo argumento y un diccionario como tercer argumento opcional. Devuelve un objeto `HttpResponse` del template renderizado con el contexto dado.

1.5.5 Levantando un error 404

Ahora veamos la view de detalle de una encuesta – la página que muestra la pregunta de una encuesta:

```
from django.http import Http404
# ...
def detail(request, poll_id):
    try:
        poll = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render(request, 'polls/detail.html', {'poll': poll})
```

El nuevo concepto aquí: la view levanta una excepción `Http404` si no existe una encuesta con ID dado.

Veremos qué podríamos poner en el template `polls/detail.html` luego, pero si quisiéramos tener el ejemplo arriba funcionando rápidamente, esto alcanza para empezar:

```
{{ poll }}
```

Un atajo: `get_object_or_404()`

Es muy común usar el método `get()` y levantar un `Http404` si el objeto no existe. Django provee un atajo. Esta la view `detail()`, actualizada:

```
from django.shortcuts import render, get_object_or_404
# ...
def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render(request, 'polls/detail.html', {'poll': poll})
```

La función `get_object_or_404()` toma un modelo Django como primer argumento y un número arbitrario de argumentos nombrados, que se pasan a la función `get()` del manager del modelo. Levanta un `Http404` si el objeto no existe.

Filosofía

Por qué usamos una función `get_object_or_404()` en lugar de manejar automáticamente la excepción `ObjectDoesNotExist` a un nivel más arriba, o hacer que la API a nivel modelo levante `Http404` en lugar de `ObjectDoesNotExist`?

Porque esto acoplaría la capa de modelos a la capa de views. Uno de los objetivos de diseño de Django es mantener bajo acoplamiento. Cierto acoplamiento controlado se introduce en el módulo `django.shortcuts`.

Existe también una función `get_list_or_404()`, que funciona como `get_object_or_404()` – excepto que usa `filter()` en lugar de `get()`. Levanta un `Http404` si la lista es vacía.

1.5.6 Escribir una view para 404 (página no encontrada)

Cuando uno levanta un `Http404` desde una view, Django carga una view especial para manejar los errores 404. Django la busca mediante la variable `handler404` en un URLconf (sólo en el URLconf raíz; seteando `handler404` en cualquier otro lugar no tendrá efecto), que es un string con sintaxis Python de puntos – el mismo formato que se usa para los callbacks en un URLconf. Una view de 404 no tiene nada especial, es una view normal.

Uno normalmente no necesita escribir views de 404. Si uno setea `handler404`, se usa por defecto la view built-in `django.views.defaults.page_not_found()`. Opcionalmente, uno puede crear un template `404.html` en la raíz de nuestro directorio de templates. La view 404 por defecto va a usar ese template para todos los errores 404 cuando `DEBUG` está en `False` (en el módulo settings). Si creás el template, agregá por lo menos algún contenido como “Page not found”.

Un par de cosas más sobre las views de 404:

- Si `DEBUG` está en `True` (en el módulo settings), entonces la view 404 no se usa (y por lo tanto, tampoco el template `404.html`) ya que en vez se muestra el traceback.
- La view de 404 también se llama si Django no encuentra una coincidencia después de chequear cada una de las expresiones regulares en el URLconf.

1.5.7 Escribir una view para 500 (error en el servidor)

De forma similar, el URLconf raíz puede definir una variable `handler500`, que apunte a una view para llamar en caso de un error en el servidor. Los errores en el servidor se dan cuando hay errores durante la ejecución del código de la view.

También deberíamos crear un template `500.html` en la raíz del directorio de templates, y agregar algún contenido como "Something went wrong".

1.5.8 Usando el sistema de templates

Volvamos a la view `detail()` de nuestra aplicación de encuestas. Dada la variable de contexto `poll`, veamos como podría lucir el template `polls/detail.html`:

```
<h1>{{ poll.question }}</h1>
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

El sistema de templates usa sintaxis de punto para acceder a los atributos de variable. En el ejemplo de `{{ poll.question }}`, Django primero hace una búsqueda de diccionario sobre el objeto `poll`. Si eso falla, intenta una búsqueda de atributo – que en este caso, funciona. Si hubiera fallado, se hubiera intentado una búsqueda por índice de lista.

Una llamada de método se da en el loop `{% for %}`: `poll.choice_set.all` se interpreta como el código Python `poll.choice_set.all()`, que devuelve un iterable de objetos `Choice` y usable para el tag `{% for %}`.

Para más detalles sobre templates, se puede ver `template guide`.

1.5.9 Borrando URLs escritas explícitamente en templates

Recordemos que cuando escribimos el link a una encuesta en el template `polls/index.html`, el link estaba parcialmente escrito "a mano":

```
<li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
```

El problema con esto, es que es una aproximación muy acoplada que hace que sea un desafío cambiar las URLs en un proyecto como muchos templates. Sin embargo, como definimos el argumento `name` en las llamadas a `url()` en el módulo `polls.urls`, podemos eliminar la dependencia de URLs fijas usando el template tag `{% url %}`:

```
<li><a href="{% url 'detail' poll.id %}">{{ poll.question }}</a></li>
```

Nota: Si `{% url 'detail' poll.id %}` (con comillas simples) no funciona, pero `{% url detail poll.id %}` (sin las comillas) sí, significa que estás usando una versión de Django < 1.5. En ese caso, se puede agregar la siguiente declaración al inicio del template:

```
{% load url from future %}
```

La forma en la que esto funciona es buscando la definición de la URL como se especificó en el módulo `polls.urls`. Uno puede ver exactamente dónde se define el nombre 'detail':

```
...
# the 'name' value as called by the {% url %} template tag
url(r'^(?P<poll_id>\d+)/$', views.detail, name='detail'),
...
```

Si uno quisiera cambiar la URL de la view de detalle de encuesta a algo distinto, tal vez algo como `polls/specifics/12/`, en lugar de hacerlo en el template (o templates), bastaría con cambiarlo en `polls/urls.py`:

```
...
# added the word 'specifics'
url(r'^specifics/(?P<poll_id>\d+)/$', views.detail, name='detail'),
...
```

1.5.10 Espacio de nombres en URLs

El proyecto de este tutorial tiene sólo una app, `polls`. En proyectos Django reales, podría haber cinco, diez, veinte o más apps. Cómo Django distingue los nombres de las URLs entre todas las apps? Por ejemplo, la app `polls` tiene una view `detail`, y podría ser que otra app para un blog en el mismo proyecto también. Cómo hace Django para saber la view de qué app usar en un template tag `{ % url %}`?

La respuesta es agregar espacios de nombres al URLconf raíz. Cambiamos el archivo `mysite/urls.py` para incluir espacios de nombres:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls', namespace="polls")),
    url(r'^admin/', include(admin.site.urls)),
)
```

Ahora cambiamos el template `polls/index.html` de:

```
<li><a href="{% url 'detail' poll.id %}">{{ poll.question }}</a></li>
```

a que apunte a la view de detalle con el espacio de nombres:

```
<li><a href="{% url 'polls:detail' poll.id %}">{{ poll.question }}</a></li>
```

Una vez que estés cómodo escribiendo views, podés pasar a la *parte 4 del tutorial* para aprender sobre procesamiento simple de forms y views genéricas.

1.6 Escribiendo tu primera Django app, parte 4

Este tutorial comienza donde dejó el *Tutorial 3*. Vamos a continuar la aplicación de encuestas concentrándonos en procesar forms simples y reducir nuestro código.

1.6.1 Escribir un form simple

Actualicemos el template de detalle de una encuesta ("polls/detail.html") del último tutorial para que contenga un elemento HTML `<form>`:

```
<h1>{{ poll.question }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' poll.id %}" method="post">
{% csrf_token %}
{% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

Un repaso rápido:

- El template de arriba muestra un radio button para cada opción de la encuesta. El `value` de cada radio es el ID asociado a cada opción de la encuesta. El `name`, es `choice"`. Esto significa que cuando alguien elige una de las opciones y envía el form, se envía `choice=3` como data del POST. Esto es Forms HTML 101.
- Establecemos como `action` del form `{ % url 'polls:vote' poll.id %}`, y `method="post"`. Usar `method="post"` (en contraposición a `method="get"`) es muy importante, porque la acción de enviar el form va a modificar datos del lado del servidor. Cada vez que uno crea un form que altere datos del lado del servidor, usar `method="post"`. Este consejo no es particular para Django; es una buena práctica de desarrollo web.
- `forloop.counter` indica cuantas veces el tag `for` iteró en el ciclo
- Como estamos creando un form POST (que puede tener el efecto de modificar datos), necesitamos preocuparnos por Cross Site Request Forgeries (CSRF). Afortunadamente no hace falta demasiado, porque Django viene con un sistema fácil de usar para protegerse contra este tipo de ataques. Simplemente todos los forms que hagan POST contra una URL interna deberían usar el template tag `{ % csrf_token %}`.

Ahora vamos a crear una view Django que maneje los datos enviados y haga algo con ellos. Recordemos que en el *Tutorial 3* creamos un URLconf para nuestra app que incluía la siguiente línea:

```
url(r'^(?P<poll_id>\d+)/vote/$', views.vote, name='vote'),
```

También habíamos creado una implementación boba de la función `vote()`. Hagamos una implementación real. Agregamos lo siguiente a `polls/views.py`:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
from polls.models import Choice, Poll
# ...
def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Vuelve a mostrar el form.
        return render(request, 'polls/detail.html', {
            'poll': p,
            'error_message': "You didn't select a choice.",
        })
```

```
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Siempre devolver un HttpResponseRedirect después de procesar
        # exitosamente el POST de un form. Esto evita que los datos se
        # puedan postear dos veces si el usuario vuelve atrás en su browser.
        return HttpResponseRedirect(reverse('polls:results', args=(p.id,)))
```

Este código incluye algunas cosas que no hemos cubierto hasta el momento:

- `request.POST` es un objeto diccionario-like que nos permite acceder a los datos enviados usando los nombres como clave. En este caso `request.POST['choice']` devuelve el ID de la opción elegida, como string. Los valores de `request.POST` son siempre strings.

  Notar que Django también provee `request.GET` para acceder a los datos en el GET de la misma manera – pero estamos usando explícitamente `request.POST` en nuestro código para asegurarnos de que los datos solamente se alteren vía una llamada POST.

- `request.POST['choice']` va a levantar `KeyError` si `choice` no estuviera en los datos del POST. El código de arriba chequea por esta excepción y en ese caso vuelve a mostrar el form con un mensaje de error.

- Después de incrementar el contador de la opción, el código devuelve un `HttpResponseRedirect` en lugar de un `HttpResponse`. `HttpResponseRedirect` toma un único argumento: la URL a la que el usuario será redirigido (ver el punto siguiente sobre cómo construir la URL en este caso).

  Como dice el comentario en el código de arriba, uno siempre debería devolver un `HttpResponseRedirect` después de manejar exitosamente un POST. Este consejo no es específico a Django; es una buena práctica de desarrollo web.

- Estamos usando la función `reverse()` en el constructor de `HttpResponseRedirect`. Esta función nos ayuda a no escribir explícitamente una URL en la función de view. Se le pasa el nombre de la view a la que queremos pasar el control y los argumentos variables del patrón de URL que apunta a esa view. En este caso, usando el URLconf que configuramos en el Tutorial 3, esta llamada a `reverse()` devolvería un string como el siguiente:

  ```
  '/polls/3/results/'
  ```

  ... donde `3` es el valor de `p.id`. Esta URL nos va a redirigir, llamando a la view `'results'` para mostrar la página final.

Como se mencionó en el Tutorial 3, `request` es un objeto `HttpRequest`. Para más detalles sobre este tipo de objetos, se puede ver la `documentación sobre request y response`.

Después de que alguien vota en una encuesta, la view `vote()` lo redirige a la página de resultados de la encuesta. Escribamos esta view:

```
def results(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render(request, 'polls/results.html', {'poll': poll})
```

Esto es casi exactamente igual a la view `detail()` del *Tutorial 3*. La única diferencia es el nombre del template. Vamos a solucionar esta redundancia luego.

Ahora, creamos el template `polls/results.html`:

```
<h1>{{ poll.question }}</h1>

<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
```

```
</ul>

<a href="{% url 'polls:detail' poll.id %}">Vote again?</a>
```

Vamos a `/polls/1/` en el browser y votamos en la encuesta. Deberíamos ver una página de resultados que se actualiza cada vez que uno vota. Si se envía el form sin elegir una opción, se debería mostrar un mensaje de error.

1.6.2 Usando views genéricas (generic views): Menos código es mejor

Las views `detail()` (del *Tutorial 3*) y `results()` son súper simples – y, cómo se menciona arriba, redundantes. La view `index()` (también del Tutorial 3), que muestra una lista de encuestas, es similar.

Estas views representan un caso común en el desarrollo web básico: obtener datos de una base de datos de acuerdo a un parámetro pasado en la URL, cargar un template y devolver el template renderizado. Por ser tan usual, Django provee un atajo, el sistema de "generic views".

Las views genéricas abstraen patrones comunes, al punto en que uno no necesita prácticamente escribir código Python en una app.

Vamos a convertir nuestra app para usar views genéricas, y poder borrar parte de nuestro código original. Son solamente unos pocos pasos:

1. Convertir el URLconf.
2. Borrar algunas de las views que teníamos, ya no necesarias.
3. Arreglar el manejo de URL para las nuevas views.

Para más detalles, seguir leyendo.

Por qué cambiar nuestro código?

Generalmente, cuando uno escribe una app Django, evaluará si usar views genéricas es una buena solución para el problema en cuestión, y las utilizará desde el comienzo, en lugar de refactorear el código a mitad de camino. Este tutorial intencionalmente se centró en escribir views "sin ayuda" hasta ahora, para aprender los conceptos principales.

Uno debería aprender matemática básica antes de empezar a usar una calculadora.

Primero, abrimos el URLconf `polls/urls.py` y lo cambiamos de la siguiente manera:

```
from django.conf.urls import patterns, url
from django.views.generic import DetailView, ListView
from polls.models import Poll

urlpatterns = patterns('',
    url(r'^$',
        ListView.as_view(
            queryset=Poll.objects.order_by('-pub_date')[:5],
            context_object_name='latest_poll_list',
            template_name='polls/index.html'),
        name='index'),
    url(r'^(?P<pk>\d+)/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/detail.html'),
        name='detail'),
    url(r'^(?P<pk>\d+)/results/$',
        DetailView.as_view(
            model=Poll,
```

```
            template_name='polls/results.html'),
        name='results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'polls.views.vote', name='vote'),
)
```

Estamos usando dos views genéricas: `ListView` y `DetailView`. Estas dos views nos abstraen de los conceptos de "mostrar una lista de objetos" y "mostrar el detalle de un objeto particular", respectivamente.

- Cada view genérica necesita saber sobre qué modelo actuar. Esto se define usando el parámetro `model`.
- La view genérica `DetailView` espera el valor de clave primaria capturado de la URL con nombre `"pk"`, entonces cambiamos `poll_id` a `pk`.

Por defecto, la view genérica `DetailView` usa un template llamado `<app name>/<model name>_detail.html`. En nuestro caso, usará el template `"polls/poll_detail.html"`. El argumento `template_name` es usado para indicarle a Django que use un template de nombre específico en lugar de usar el nombre de template autogenerado por defecto. También especificamos `template_name` para la view `results` – esto nos asegura que la view de resultados y la de detalle tiene un aspecto diferente al renderizarse, aún cuando ambas usan `DetailView` por detrás.

De forma similar, la view genérica `ListView` usa un template por defecto llamado `<app name>/<model name>_list.html`; usamos `template_name` para indicarle a `ListView` que use el template ya existente `"polls/index.html"`.

En partes anteriores de este tutorial, los templates recibían un contexto que contenía las variables `poll` y `latest_poll_list`. Para `DetailView` la variable `poll` es provista automáticamente – como estamos usando un modelo Django (`Poll`), Django puede determinar un nombre adecuado para la variabl de contexto. Sin embargo, para ListView, el nombre de variable de contexto generado automáticamente es `poll_list`. Para sobreescribir este valor, pasamos la opción `context_object_name`, especificando que queremos usar `latest_poll_list` como nombre. Otra alternativa sería cambiar los templates para adecuarlos a los nombres por defecto – pero es mucho más fácil decirle a Django que use el nombre de variable que queremos.

Ahora podemos borrar las views `index()`, `detail()` y `results()` de `polls/views.py`. No las necesitamos más – fueron reemplazadas por views genéricas.

Corremos el servidor, y usamos nuestra app, ahora basada en views genéricas.

Para más detalles sobre views genéricas, se puede ver la `documentación sobre views genéricas`.

Una vez que estés cómodo con forms y views genéricas, podés leer la *parte 5 de este tutorial* para aprender sobre cómo testear nuestra app de encuestas.

1.7 Escribiendo tu primera Django app, parte 5

Este tutorial comienza donde dejó el *Tutorial 4*. Hemos construido una aplicación de encuestas, y ahora vamos a agregar algunos tests automáticos para la misma.

1.7.1 Introduciendo testing automatizado

Qué son los tests automatizados?

Los tests son rutinas simples que chequean el funcionamiento de nuestro código.

Afectan diferentes niveles. Algunos tests pueden aplicar a un detalle menor - *un método particular de un modelo devuelve el valor esperado?*, mientras otros verifican el funcionamiento general del software - *una secuencia de entradas del usuario producen el resultado deseado?* No difiere del tipo de pruebas que uno hacía en el *Tutorial 1*, usando el

shell para examinar el comportamiento de un método, o correr la aplicación e ingresar datos para chequear cómo se comporta.

Lo que diferencia a los tests *automatizados* es que el trabajo de hacer las pruebas lo hace el sistema por uno. Uno crea un conjunto de tests una vez, y luego a medida que se hacen cambios a la app, se puede verificar que el código todavía funciona como estaba originalmente pensado, sin tener que usar tiempo para hacer testing manual.

Por qué es necesario tener tests

Por qué crear tests, y por qué ahora?

Uno podría pensar que ya tiene suficiente con ir aprendiendo Python/Django, y todavía tener que aprender algo más puede parecer demasiado y quizás innecesario. Después de todo nuestra aplicación de encuestas está funcionando; tomarse el trabajo de escribir tests automatizados no va a hacer que funcione mejor. Si crear esta aplicación de encuestas es la última tarea de programación con Django que vas a hacer, es cierto, no necesitás saber cómo crear tests automatizados. Pero, si no es el caso, este es un excelente momento para aprenderlo.

Los tests nos ahorrarán tiempo

Hasta cierto punto, 'chequear que todo parece funcionar' es un test satisfactorio. En una aplicación más sofisticada, uno podría tener docenas de interacciones complejas entre componentes.

Un cambio en cualquiera de esos componentes podría tener consecuencias inesperadas en el comportamiento de la aplicación. Chequear que todavía 'parece funcionar' podría significar recorrer el funcionamiento del código con veinte variaciones diferentes de pruebas para estar seguros de que no se rompió nada - no es un buen uso del tiempo.

Esto es particularmente cierto cuando tests automatizados podrían hacerlo por uno en segundos. Si algo se rompió, los tests también ayudan a identificar el código que está causando el comportamiento inesperado.

Algunas veces puede parecer una tarea que nos distrae de nuestro creativo trabajo de programación para dedicarnos al poco atractivo asunto de escribir tests, especialmente cuando uno sabe que el código está funcionando correctamente.

Sin embargo, la tarea de escribir tests rinde mucho más que gastar horas probando la aplicación manualmente o intentando identificar la causa de un problema que se haya producido con un cambio.

Los tests no sólo identifican problemas, los previenen

Es un error pensar que los tests son un aspecto negativo del desarrollo.

Sin tests, el propósito o comportamiento esperado de una aplicación podría ser poco claro. Incluso siendo código propio, algunas veces uno se encuentra tratando de adivinar qué es lo que hacía exactamente.

Los tests cambian eso; iluminan el código desde adentro, y cuando algo va mal, iluminan la parte que va mal - *aún si uno no se dio cuenta de que algo va mal*.

Los tests hacen el código más atractivo

Uno podría crear una obra de software brillante, pero muchos desarrolladores simplemente se van a rehusar de verla porque no tiene tests; sin tests, no van a confiar en ese software. Jacob Kaplan-Moss, uno de los desarrolladores originales de Django, dice "El código sin tests está roto por diseño".

El hecho de que otros desarrolladores quieran ver tests en nuestro software antes de tomarlo seriamente es otra razón para empezar a escribir tests.

Los tests ayudan a trabajar a un equipo

Los puntos anteriores están escritos desde el punto de vista de un único desarrollador manteniendo una aplicación. Aplicaciones complejas son mantenidas por equipos. Los tests garantizan que otros colegas no rompan nuestro código sin darse cuenta (y que uno no rompe el de ellos sin saberlo). Si uno quiere vivir de la programación con Django, debe ser bueno escribiendo tests!

1.7.2 Estrategias de testing básicas

Hay varias maneras de aprender a escribir tests.

Algunos programadores siguen una disciplina llamada "test-driven development" (desarrollo dirigido por tests); los tests se escriben antes de escribir el código. Puede parecer contra intuitivo, pero en realidad es similar a lo que la mayoría de la gente haría: describir un problema, luego crear el código que lo resuelve. Test-driven development simplemente formaliza el problema como un caso de test Python.

A menudo, alguien nuevo en testing va a crear código y más tarde decidir que debería tener algunos tests. Tal vez hubiera sido mejor escribir los tests antes, pero nunca es tarde para empezar.

A veces es difícil darse cuenta por dónde empezar al escribir tests. Si uno escribió varios miles de líneas de Python, elegir qué testear puede no ser fácil. En ese caso, es provechoso escribir el primer test la próxima vez que uno hace un cambio, ya sea una nueva funcionalidad o un fix de un bug.

Vamos a hacer eso entonces.

1.7.3 Escribiendo nuestro primer test

Identificamos un bug

Por suerte, hay un pequeño bug en la aplicación `polls` que podemos arreglar: el método `Poll.was_published_recently()` devuelve `True` si una `Poll` fue publicada el día anterior (que está bien), pero también si el campo `pub_date` es en el futuro (que no está bien!).

Podemos verlo en el Admin; creamos una encuesta cuya fecha es en el futuro; veremos que el listado de encuestas nos dice que fue publicada recientemente.

También podemos verlo usando el shell:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Poll
>>> # creamos una instancia de Poll con pub_date 30 días en el futuro
>>> future_poll = Poll(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # se publicó recientemente?
>>> future_poll.was_published_recently()
True
```

Dado que las cosas en el futuro no son 'recientes', esto es incorrecto.

Creamos un test que exponga el bug

Lo que acabamos de hacer en el shell para verificar el problema es exactamente lo que podemos hacer en un test automatizado. Hagámoslo entonces.

El mejor lugar para los tests de una aplicación es el archivo `tests.py` - el corredor de tests va a buscar los tests allí automáticamente.

Ponemos el siguiente código en el archivo `tests.py` en la aplicación `polls` (vamos a notar que ya contiene algunos tests de ejemplo, los borramos):

```
import datetime

from django.utils import timezone
from django.test import TestCase

from polls.models import Poll

class PollMethodTests(TestCase):

    def test_was_published_recently_with_future_poll(self):
        """
        was_published_recently() should return False for polls whose
        pub_date is in the future
        """
        future_poll = Poll(pub_date=timezone.now() + datetime.timedelta(days=30))
        self.assertEqual(future_poll.was_published_recently(), False)
```

Lo que hemos hecho aquí es crear una subclase de `django.test.TestCase` con un método que crea una instancia de `Poll` con un valor de `pub_date` en el futuro. Luego chequeamos la salida de `was_published_recently()` - que *debería* ser False.

Corriendo los tests

En la terminal, podemos correr nuestro test:

```
python manage.py test polls
```

y veremos algo como:

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_poll (polls.tests.PollMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_poll
    self.assertEqual(future_poll.was_published_recently(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Qué paso aquí:

- `python manage.py test polls` buscó tests en la aplicación `polls`
- encontró una subclase de `django.test.TestCase`
- creó una base de datos especial para los tests
- buscó los métodos de test - aquellos cuyo nombre comienza con `test`
- en `test_was_published_recently_with_future_poll` creó una instancia de `Poll` cuyo valor para el campo `pub_date` es 30 días en el futuro

- ... y usando el método `assertEqual()`, descubrió que `was_published_recently()` devuelve `True`, a pesar de que queríamos que devolviera `False`

La corrida nos informa qué test falló e incluso la línea en la que se produjo la falla.

Arreglando el bug

Ya sabemos cuál es el problema: `Poll.was_published_recently()` debería devolver `False` si `pub_date` tiene un valor en el futuro. Corregimos el método en `models.py`, para que sólo devuelva `True` si además la fecha es en el pasado:

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <  now
```

y corremos el test nuevamente:

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

Después de identificar un bug, escribimos el test que lo expone y corregimos el problema en el código para que nuestro test pase.

Muchas cosas pueden salir mal con nuestra aplicación en el futuro, pero podemos estar seguros de que no vamos a re-introducir este bug inadvertidamente, porque sólo correr el test nos lo haría notar inmediatemente. Podemos considerar esta porción de nuestra aplicación funcionando y cubierta a futuro.

Tests más exhaustivos

Mientras estamos aquí, podemos mejorar la cobertura del método `was_published_recently()`; de hecho, sería vergonzoso si arreglando un bug hubiéramos introducido uno nuevo.

Agregamos dos métodos más a la misma clase, para testear el comportamiento del método de forma más exhaustiva:

```
def test_was_published_recently_with_old_poll(self):
    """
    was_published_recently() should return False for polls whose pub_date
    is older than 1 day
    """
    old_poll = Poll(pub_date=timezone.now() - datetime.timedelta(days=30))
    self.assertEqual(old_poll.was_published_recently(), False)

def test_was_published_recently_with_recent_poll(self):
    """
    was_published_recently() should return True for polls whose pub_date
    is within the last day
    """
    recent_poll = Poll(pub_date=timezone.now() - datetime.timedelta(hours=1))
    self.assertEqual(recent_poll.was_published_recently(), True)
```

Y ahora tenemos tres tests que confirman que `Poll.was_published_recently()` devuelve valores sensatos para encuestas pasadas, recientes y futuras.

De nuevo, `polls` es una aplicación simple, pero sin importar lo complejo que pueda crecer en el futuro o la interacción que pueda tener con otro código, tenemos alguna garantía de que el método para el cual hemos escrito tests se comportará de la manera esperada.

1.7.4 Testeando una view

La aplicación polls no discrimina: va a publicar cualquier encuesta, incluyendo aquellas cuyo campo `pub_date` tiene un valor en el futuro. Deberíamos mejorar esto. Tener un `pub_date` en el futuro debería significar que la encuesta se publica en ese momento, pero permanece invisible hasta entonces.

Un test para una view

Cuando arreglamos el bug de arriba, escribimos un test primero y luego el código que lo arreglaba. De hecho fue un ejemplo simple de test-driven development, pero no importa en realidad el orden en que lo hicimos.

En nuestro primer test nos concentramos en el comportamiento interno del código. Para este test, queremos chequear el comportamiento como lo experimentaría un usuario mediante el browser.

Antes de intentar arreglar cualquier cosa, veamos las herramientas a nuestra disposición.

El cliente para test de Django

Django provee un cliente para test, `Client`, para simular la interacción del usuario con el código a nivel view. Podemos usarlo en `tests.py` o incluso en el shell.

Empezaremos de nuevo con el shell, donde necesitamos hacer un par de cosas que no serán necesarias en `tests.py`. La primera es crear el ambiente de test en el shell:

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

A continuación necesitamos importar la clase del cliente para test (luego en `tests.py` vamos a usar la clase `django.test.TestCase`, que viene con su propio cliente, así que este paso no será requerido):

```
>>> from django.test.client import Client
>>> # creamos una instancia del cliente para nuestro uso
>>> client = Client()
```

Con eso listo, podemos pedirle al cliente que haga trabajo por nosotros:

```
>>> # obtener la respuesta para '/'
>>> response = client.get('/')
>>> # deberíamos esperar un 404 de esa dirección
>>> response.status_code
404
>>> # por otro lado deberíamos esperar encontrar algo en '/polls/'
>>> # vamos a usar 'reverse()' en lugar de escribir explícitamente la URL
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
'\n\n\n    <p>No polls are available.</p>\n\n'
>>> # notar - se pueden obtener resultados inesperados si el ''TIME_ZONE''
>>> # en ''settings.py'' no es correcto. Si necesitaras cambiarlo,
>>> # también tendrás que reiniciar la sesión en el shell
```

```
>>> from polls.models import Poll
>>> from django.utils import timezone
>>> # creamos una encuesta y la guardamos
>>> p = Poll(question="Who is your favorite Beatle?", pub_date=timezone.now())
>>> p.save()
>>> # chequeamos la respuesta una vez más
>>> response = client.get('/polls/')
>>> response.content
'\n\n\n    <ul>\n    \n        <li><a href="/polls/1/">Who is your favorite Beatle?</a></li>\n    \n
>>> response.context['latest_poll_list']
[<Poll: Who is your favorite Beatle?>]
```

Mejorando nuestra view

La lista de encuestas nos muestra entradas que no están publicadas todavía (i.e. aquellas que tienen `pub_date` en el futuro). Arreglémoslo.

En el *Tutorial 4* reemplazamos las funciones de view en `views.py` por `ListView` en `urls.py`:

```
url(r'^$',
    ListView.as_view(
        queryset=Poll.objects.order_by('-pub_date')[:5],
        context_object_name='latest_poll_list',
        template_name='polls/index.html'),
    name='index'),
```

`response.context_data['latest_poll_list']` extrae los datos que la view pone en el contexto.

Necesitamos corregir la línea que nos da el `queryset`:

```
queryset=Poll.objects.order_by('-pub_date')[:5],
```

Vamos a cambiar el queryset para que también chequee la fecha comparándola con `timezone.now()`. Primero necesitamos agregar un import:

```
from django.utils import timezone
```

y luego debemos corregir la función `url` existente:

```
url(r'^$',
    ListView.as_view(
        queryset=Poll.objects.filter(pub_date__lte=timezone.now) \
            .order_by('-pub_date')[:5],
        context_object_name='latest_poll_list',
        template_name='polls/index.html'),
    name='index'),
```

`Poll.objects.filter(pub_date__lte=timezone.now)` devuelve un queryset que contiene las instancias de `Poll` cuyo campo `pub_date` es menor o igual que - esto es, anterior o igual a - `timezone.now`. Notar que usamos un `callable` (en este caso, el nombre de una función) como argumento del queryset, `timezone.now`, que se evaluará al momento del pedido. Si hubiéramos incluido los paréntesis, `timezone.now()` se hubiera evaluado solamente una vez, cuando arrancamos el servidor web.

Testeando nuestra nueva view

Ahora podemos verificar que se comporta como esperamos levantando el servidor de desarrollo, cargando el sitio en el browser, creando `Polls` con fechas en el pasado y en el futuro, y chequeando que solamente se listan aquellas

que han sido publicadas. Uno no quiere repetir estos pasos *cada vez que se hace un cambio que podría afectar esto* - vamos a crear entonces un test, basándonos en nuestra sesión con el shell.

Agregamos lo siguiente a `polls/tests.py`:

```
from django.core.urlresolvers import reverse
```

vamos a crear un método `factory` para crear encuestas, como así también una nueva clase de test:

```
def create_poll(question, days):
    """
    Creates a poll with the given `question` published the given number of
    `days` offset to now (negative for polls published in the past,
    positive for polls that have yet to be published).
    """
    return Poll.objects.create(question=question,
        pub_date=timezone.now() + datetime.timedelta(days=days))

class PollViewTests(TestCase):
    def test_index_view_with_no_polls(self):
        """
        If no polls exist, an appropriate message should be displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_poll_list'], [])

    def test_index_view_with_a_past_poll(self):
        """
        Polls with a pub_date in the past should be displayed on the index page.
        """
        create_poll(question="Past poll.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_poll_list'],
            ['<Poll: Past poll.>']
        )

    def test_index_view_with_a_future_poll(self):
        """
        Polls with a pub_date in the future should not be displayed on the
        index page.
        """
        create_poll(question="Future poll.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.", status_code=200)
        self.assertQuerysetEqual(response.context['latest_poll_list'], [])

    def test_index_view_with_future_poll_and_past_poll(self):
        """
        Even if both past and future polls exist, only past polls should be
        displayed.
        """
        create_poll(question="Past poll.", days=-30)
        create_poll(question="Future poll.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_poll_list'],
```

```
            ['<Poll: Past poll.>']
        )

    def test_index_view_with_two_past_polls(self):
        """
        The polls index page may display multiple polls.
        """
        create_poll(question="Past poll 1.", days=-30)
        create_poll(question="Past poll 2.", days=-5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_poll_list'],
             ['<Poll: Past poll 2.>', '<Poll: Past poll 1.>']
        )
```

Vamos a mirarlo más detenidamente.

Primero tenemos un método, `create_poll`, para evitar la repetición en el proceso de crear encuestas.

`test_index_view_with_no_polls` no crea encuestas, pero chequea el mensaje "No polls are available." y verifica que `latest_poll_list` es vacío. Notar que la clase `django.test.TestCase` provee algunos métodos de aserción adicionales. En estos ejemplos usamos `assertContains()` y `assertQuerysetEqual()`.

En `test_index_view_with_a_past_poll`, creamos una encuestas y verificamos que aparece en el listado.

En `test_index_view_with_a_future_poll`, creamos una encuesta con `pub_date` en el futuro. La base de datos se resetea para cada método de test, entonces la primera encuesta no está más, y entonces nuevamente no deberíamos tener ninguna entrada en el listado.

Y así sucesivamente. En efecto, estamos usando los tests para contar una historia de entrada de encuestas y la experiencia del usuario en el sitio, y chequeando que en cada estado y para cada cambio en el estado del sistema, se publican los resultados esperados.

Testeando `DetailView`

Lo que tenemos funciona bien; sin embargo, aunque las encuestas futuras no aparecen en el *index*, un usuario todavía puede verlas si saben o adivinan la URL correcta. Necesitamos restricciones similares para `DetailViews`, para lo cual agregamos:

```
queryset=Poll.objects.filter(pub_date__lte=timezone.now)
```

por ejemplo:

```
url(r'^(?P<pk>\d+)/$',
    DetailView.as_view(
        queryset=Poll.objects.filter(pub_date__lte=timezone.now),
        model=Poll,
        template_name='polls/detail.html'),
    name='detail'),
```

y por supuesto, vamos a agregar algunos tests para chequear que una instancia de `Poll` con fecha `pub_date` en el pasado se puede mostrar, y que una con fecha en el futuro no:

```
class PollIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_poll(self):
        """
        The detail view of a poll with a pub_date in the future should
        return a 404 not found.
        """
```

```
        future_poll = create_poll(question='Future poll.', days=5)
        response = self.client.get(reverse('polls:detail', args=(future_poll.id,)))
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_poll(self):
        """
        The detail view of a poll with a pub_date in the past should display
        the poll's question.
        """
        past_poll = create_poll(question='Past Poll.', days=-5)
        response = self.client.get(reverse('polls:detail', args=(past_poll.id,)))
        self.assertContains(response, past_poll.question, status_code=200)
```

Ideas para más tests

Deberíamos agregar un argumento `queryset` similar para la otra URL de `DetailView`, y crear una nueva clase de test para esta view. Sería parecido a lo que hemos hecho ya; de hecho, habría bastante repetición.

Podríamos también mejorar nuestra aplicación, agregando tests en el camino. Por ejemplo, es tonto permitir que encuestas sin opciones se puedan publicar en el sitio. Entonces, nuestras views podrían chequear esto, y excluir esas encuestas. Los tests crearían una instancia de `Poll` sin `Choices` relacionados, y luego verificarían que no se publica, como así también que si se crea una instancia de `Poll` *con* `Choices`, se verifica que *sí* se publica.

Quizás usuarios administradores logueados deberían poder ver encuestas no publicadas, pero no los demás usuarios. Una vez más: cualquier funcionalidad que necesite agregarse debe estar acompañada por tests, ya sea escribiendo el test primero y luego el código que lo hace pasar, o escribir el código de la funcionalidad primero y luego escribir el test para probarla.

En cierto punto uno mira sus tests y se pregunta si el código de los tests no está creciendo demasiado, lo que nos lleva a:

1.7.5 Tratándose de tests, más es mejor

Puede parecer que nuestros tests están creciendo fuera de control. A este ritmo pronto tendremos más código en nuestros tests que en nuestra aplicación, y la repetición no es estética, comparada con lo conciso y elegante del resto de nuestro código.

No importa. Dejémoslos crecer. En gran medida, uno escribe un test una vez y luego se olvida. Va a seguir cumpliendo su función mientras uno continúa desarrollando su programa.

Algunas veces los tests van a necesitar actualizarse. Supongamos que corregimos nuestras views para que solamente se publiquen `Polls` con `Choices`. En ese caso, muchos de nuestros tests existentes van a fallar - *diciéndonos qué tests actualizar y corregir*, así que hasta cierto punto los tests pueden cuidarse ellos mismos.

A lo sumo, mientras uno continúa desarrollando, se puede encontrar que hay algunos tests que se hacen redundantes. Incluso esto no es un problema; en testing, la redundancia es algo *bueno*.

Mientras que los tests estén organizados de manera razonable, no se van a hacer inmanejables. Algunas buenas prácticas:

- un `TestClass` separado para cada modelo o view
- un método de test separado para cada conjunto de condiciones a probar
- nombres de método de test que describan su función

1.7.6 Testing adicional

Este tutorial solamente presenta lo básico sobre testing. Hay bastante más que se puede hacer, y existen herramientas muy útiles a disposición para lograr cosas muy interesantes.

Por ejemplo, mientras que nuestros tests han cubierto la lógica interna de un modelo y la forma en que nuestras views publican información, uno podría usar un framework "in-browser" como Selenium para testear la manera en que el HTML se renderiza en un browser. Estas herramientas nos permiten no sólo chequear el comportamiento de nuestro código Django, si no también, por ejemplo, nuestro JavaScript. Es algo muy curioso ver los tests ejecutar un browser y empezar a interactuar con nuestro sitio como si un humano lo estuviera controlando! Django incluye `LiveServerTestCase` para facilitar la integración con herramientas como Selenium.

Si uno tiene una aplicación compleja, podría querer correr los tests automáticamente con cada commit con el propósito de 'integración continua'_, de tal manera de automatizar - al menos parcialmente - el control de calidad.

Una buena forma de encontrar partes de nuestra aplicación sin testear es chequar el cubrimiento del código. Esto también ayuda a identificar código frágil o muerto. Si uno no puede testear un fragmento de código, en general significa que ese código debería refactorearse o borrarse. Ver *Integración con coverage* para más detalles.

`Testeando aplicaciones Django` tiene información exhaustiva sobre testing.

1.7.7 Qué sigue?

El tutorial para principiantes termina aquí. A continuación podrías querer chequar algunos punteros sobre cómo seguir desde aquí.

Si estás familiarizado con empaquetar Python y estás interesado en aprender cómo convertir `polls` en una "app reusable", chequeá *Tutoral avanzado: Cómo escribir apps reusables*.

1.8 Tutorial avanzado: Cómo escribir apps reusables

Este tutorial avanzado comienza donde dejó el *Tutorial 5*. Vamos a convertir nuestra app en un paquete Python standalone de tal manera que se pueda reusar en nuevos proyectos y compartir con otra gente.

Si todavía no completaste los Tutoriales 1-5, te alentamos a hacerlo, ya que nos basaremos en ese proyecto durante este tutorial.

1.8.1 La reusabilidad importa

Es mucho trabajo diseñar, construir, testear y mantener una aplicación web. Varios proyectos Python y Django comparten problemas comunes. No sería bueno poder ahorrarse algo de este trabajo repetido?

Reusabilidad es el estilo de vida en Python. The Python Package Index (PyPI) tiene un amplio abanico de paquetes que uno puede usar en sus propios programas Python. También vale la pena chequear Django Packages para apps reusables que se pueden incorporar en nuestros proyectos. Django mismo es también un paquete Python. Esto significa que uno puede partir de paquetes Python o Django apps existentes y componerlos en un proyecto web propio. Solamente es necesario escribir las partes que hacen nuestro proyecto único.

Digamos que estamos por comenzar un nuevo proyecto que necesita una app de encuestas como la que hemos estado desarrollando. Cómo hacemos que sea reusable? Afortunadamente, estamos en el buen camino. En el *Tutorial 3* vimos como desacoplar la app `polls` del URLconf a nivel proyecto usando `include`. En este tutorial vamos a ir más allá para lograr que nuestra app sea fácil de usar en nuevos proyectos y quede lista para publicarla y que otros puedan instalarla y usarla.

Paquete? App?

Un paquete Python provee una manera de agrupar código Python relacionado para facilitar su reuso. Un paquete contiene uno o más archivos de código Python (también conocidos como "módulos").

Un paquete se puede importar con `import foo.bar` o `from foo import bar`. Para que un directorio (como `polls`) sea un paquete, debe contener un archivo especial, `__init__.py`, que incluso puede estar vacío.

Una *app* Django es sólo un paquete Python que está pensado específicamente para usarse en un proyecto Django. Una app puede también usar algunas convenciones comunes de Django, como tener un archivo `models.py`.

Más adelante usamos el término *empaquetar* para describir el proceso de hacer que un paquete Python sea fácil de instalar para otros. Puede resultar un poco confuso, lo sabemos.

1.8.2 Completando la reusabilidad de nuestra app

Después de los tutoriales anteriores, nuestro proyecto debería verse así:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    polls/
        admin.py
        __init__.py
        models.py
        tests.py
        urls.py
        views.py
```

Podríamos tener también un directorio en algún lado llamado `mytemplates` creado durante el *Tutorial 2*. Especificamos esta ubicación en el setting TEMPLATE_DIRS. Este directorio debería verse así:

```
mytemplates/
    admin/
        base_site.html
    polls/
        detail.html
        index.html
        results.html
```

La app `polls` ya es un paquete Python, gracias al archivo `polls/__init__.py`. Es un buen comienzo, pero todavía no podemos agarrar este paquete y usarlo en un proyecto nuevo. Los templates de esta app están guardados en el directorio `mytemplates` a nivel proyecto. Para que la app sea autocontenida, debe también contener los templates necesarios.

Dentro de la app `polls` creamos un nuevo directorio `templates`. Movemos el directorio `polls` de templates desde `mytemplates` al nuevo directorio `templates` que acabamos de crear. Ahora nuestro proyecto debería verse así:

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

```
        urls.py
        wsgi.py
    polls/
        admin.py
        __init__.py
        models.py
        templates/
            polls/
                detail.html
                index.html
                results.html
        tests.py
        urls.py
        views.py
```

Mientras que el directorio de templates a nivel proyecto se vería así:

```
mytemplates/
    admin/
        base_site.html
```

Muy bien! Es un buen momento para confirmar que nuestra aplicación `polls` todavía funciona correctamente. Cómo hace Django para saber dónde encontrar los templates de `polls` aún cuando no modificamos `TEMPLATE_DIRS`? Django tiene un setting `TEMPLATE_LOADERS` que contiene una lista de callables que saben como importar los templates de diferentes lugares. Uno de los cargadores por defecto es `django.template.loaders.app_directories.Loader` que busca un subdirectorio "templates" en cada una de las apps en `INSTALLED_APPS`.

Ahora el directorio `polls` podría copiarse en un nuevo proyecto Django y ser reusado inmediatemente. Todavía no está listo para publicarse, sin embargo. Para ello necesitamos empaquetar la app y hacer fácil su instalación para otros.

Por qué el anidamiento?

Por qué creamos un directorio `polls` dentro de `templates` si ya estamos dentro de la app polls? Este directorio se agrega para evitar conflictos con el cargador de templates de `app_directories`. Por ejemplo, si dos apps tuvieran un template llamado `base.html`, sin este directorio extra no sería posible distinguir entre los dos. Es una buena convención usar el nombre de la app para este directorio.

1.8.3 Instalando algunos prerrequisitos

El estado actual del empaquetado en Python está un poco confuso con varias herramientas. Para este tutorial vamos a usar distribute para construir nuestro paquete. Es un fork mantenido por la comunidad de un proyecto anterior, `setuptools`. Vamos a usar también pip para desinstalarlo una vez que terminemos. Deberías instalar estos dos paquetes ahora. Si necesitaras ayuda, podés chequear *cómo instalar Django con pip*. De la misma manera se puede instalar `distribute`.

1.8.4 Empaquetando nuestra app

El *empaquetar* Python se refiere a preparar nuestra app en un formato específico que pueda ser fácilmente instalable para su uso. Django mismo viene empaquetado de forma muy similar. Para una app pequeña como `polls`, el proceso no es muy complicado.

1. Primero, creamos un directorio padre para `polls`, fuera del proyecto Django. Llamaremos a este directorio `django-polls`.

Eligiendo un nombre para nuestra app

Cuando uno elige un nombre para un paquete es buena idea chequear fuentes como PyPI para evitar conflictos de nombre con paquetes existentes. A menudo es útil usar el prefijo `django-` para el nombre cuando uno crea un paquete para distruirlo. Esto ayuda a que aquellos que buscan apps de Django identifiquen la app como específica para Django.

2. Movemos el directorio `polls` dentro de `django-polls`.
3. Creamos un archivo `django-polls/README.txt` con el siguiente contenido:

```
=====
Polls
=====

Polls es una app Django simple para encuestas. Por cada pregunta, los
visitantes pueden elegir entre un número fijo de respuestas.

La documentación detallada está en el directorio "docs".

Comienzo rápido
---------------

1. Agregar "polls" al setting INSTALLED_APPS::

      INSTALLED_APPS = (
          ...
          'polls',
      )

2. Incuir el URLconf de polls en el urls.py del proyecto::

      url(r'^polls/', include('polls.urls')),

3. Correr `python manage.py syncdb` para crear los modelos de polls.

4. Levantar el servidor de desarrollo y visitar http://127.0.0.1:8000/admin/
  para crear una encuesta (es necesario tener la app Admin habilitada).

5. Visitar http://127.0.0.1:8000/polls/ para participar en una encuesta.
```

(se puede ver la versión original del README en inglés).

4. Creamos un archivo `django-polls/LICENSE`. Elegir una licencia está fuera del alcance de este tutorial, pero basta decir que un código liberado públicamente sin una licencia es *inservible*. Django y muchas apps compatibles se distribuyen bajo la licencia BSD; sin embargo, uno es libre de elegir su propia licencia. Hay que tener en cuenta que esta elección afectará quién puede usar nuestro código.
5. Luego vamos a crear un archivo `setup.py` que provee los detalles sobre cómo construir e instalar la app. Escapa a este tutorial una explicación más detallada sobre este archivo, pero la documentación de distribute tiene una buena explicación. Creamos el archivo `django-polls/setup.py` con el siguiente contenido:

```
import os
from setuptools import setup

README = open(os.path.join(os.path.dirname(__file__), 'README.txt')).read()

# allow setup.py to be run from any path
```

```
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__), os.pardir)))

setup(
    name = 'django-polls',
    version = '0.1',
    packages = ['polls'],
    include_package_data = True,
    license = 'BSD License', # example license
    description = 'A simple Django app to conduct Web-based polls.',
    long_description = README,
    url = 'http://www.example.com/',
    author = 'Your Name',
    author_email = 'yourname@example.com',
    classifiers = [
        'Environment :: Web Environment',
        'Framework :: Django',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License', # example license
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Programming Language :: Python :: 2.6',
        'Programming Language :: Python :: 2.7',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)
```

Creí que íbamos a usar `distribute`?

Distribute es el reemplazo de `setuptools`. Aunque importamos de `setuptools`, como hemos instalado `distribute`, éste va a sobreescribir el import.

6. Solamente módulos y paquetes Python se incluyen por defecto en el paquete. Para incluir archivos adicionales, necesitamos crear un archivo `MANIFEST.in`. La documentación sobre distribute referida en el paso anterior revisa este archivo en más detalle. Para incluir los templates y nuestro archivo LICENSE, creamos el archivo `django-polls/MANIFEST.in` con el siguiente contenido:

   ```
   include LICENSE
   recursive-include polls/templates *
   ```

7. Es opcional, pero se recomienda, incluir documentación detallada con una app. Creamos un directorio vacío `django-polls/docs` para la futura documentación. Agregamos una línea adicional al archivo `django-polls/MANIFEST.in`:

   ```
   recursive-include docs *
   ```

   Notemos que el directorio `docs` no se va a incluir en nuestro paquete a menos que agreguemos algunos archivos dentro. Muchas apps Django también proveen su documentación de forma online mediante sitios como readthedocs.org.

8. Intentemos construir nuestro paquete con `python setup.py sdist` (corriendo desde dentro de `django-polls`). Esto crea un directorio llamado `dist` y crea un nuevo paquete, `django-polls-0.1.tar.gz`.

Para más información sobre empaquetado, se puede consultar The Hitchhiker's Guide to Packaging.

1.8.5 Usando nuestro paquete

Como movimos el directorio `polls` fuera de nuestro proyecto, no funciona más. Vamos a solucionar esto instalando nuestro nuevo paquete, `django-polls`.

Instalando como librería de usuario

Los siguientes pasos instalan `django-polls` como librería de usuario. Instalaciones de usuario tienen ventajas sobre instalar un paquete a nivel sistema, como ser usables en sistemas donde uno no tiene acceso de administrador como así también evitar que un paquete afecte servicios del sistema o a otros usuarios. Python 2.6 agregó soporte para librerías de usuario, no va a funcionar para versiones anteriores. De cualquier manera, Django 1.5 requiere Python 2.6 o superior.

Notar que las instalaciones por usuario pueden incluso afectar herramientas del sistema que corren bajo ese usuario, entonces `virtualenv` es un solución más robusta (ver más abajo).

---

1. Dentro de `django-polls/dist`, hacer untar del nuevo paquete `django-polls-0.1.tar.gz` (e.g. `tar xzvf django-polls-0.1.tar.gz`). Si usamos Windows, se puede bajar la herramienta de línea de comandos bsdtar para hacer esto, o usar una herramienta GUI como 7-zip.
2. Nos movemos al directorio creado en el paso 1 (e.g. `cd django-polls-0.1`).
3. Si usamos GNU/Linux, Mac OS X o algún otro sabor de Unix, corremos el comando `python setup.py install --user` en el prompt del shell. Si usamos Windows, abrimos un shell y corremos el comando `setup.py install --user`.

   Con suerte, nuestro proyecto Django debería funcionar correctamente de nuevo. Levantamos el servidor de desarrollo para confirmarlo.
4. Para desinstalar el paquete, usamos pip (ya *lo instalamos*, no?:

   ```
   pip uninstall django-polls
   ```

1.8.6 Publicando nuestra app

Ahora que ya empaquetamos y testeamos `django-polls`, está lista para compartirla con el mundo! Si este no fuera sólo un ejemplo, uno podría:

- Enviar el paquete vía email a un amigo.
- Subir el paquete al sitio web propio.
- Postear el paquete en un repositorio público, como The Python Package Index (PyPI).

Para más información sobre PyPI, ver la sección Quickstart de The Hitchhiker's Guide to Packaging. Un detalle que menciona esta guía es elegir la licencia bajo la cual distribuiremos nuestro código.

1.8.7 Instalando paquetes Python con virtualenv

Instalar la app polls como librería de usuario tiene algunas desventajas:

- Modificar las librerías de usuario puede afectar otro software Python de nuestro sistema.
- No podemos correr múltiples versiones de nuestro paquete (u otros con el mismo nombre).

Típicamente, estas situaciones sólo se presentan si uno mantiene varios proyectos Django. En ese caso, la mejor solución es usar virtualenv. Esta herramienta permite mantener múltiples ambientes Python aislados, cada uno con su propia copia de librerías y paquetes.

1.9 Escribiendo nuestro primer patch para Django

1.9.1 Introducción

Interesado en devolver algo a la comunidad? Quizás hayas encontrado un bug en Django que te gustaría ver arreglado, o quizás hay alguna funcionalidad que te gustaría ver incorporada.

Constribuir con Django es la mejor manera de ver nuestras preocupaciones tenidas en cuenta. Puede parece intimidante al principio, pero es realmente simple. Vamos a recorrer el proceso completo para aprender mediante un ejemplo.

Para quién es este tutorial?

Para este tutorial esperamos que tengas al menos un conocimiento básico sobre cómo funciona Django. Esto significa que deberías estar cómodo recorriendo los tutoriales existentes, *Escribiendo tu primera Django app*. Además, deberías tener una buena base de Python. Si no es el caso, Dive Into Python es un libro online fantástico (y gratis) para aquellos que comienzan con Python.

Para aquellos que no estén familiarizados con sistemas de control de versiones y Trac, van a encontrar en este tutorial y sus links la información suficiente para empezar. Sin embargo, probablemente quieras empezar a leer más sobre estas diferentes herramientas si tu plan es contribuir con Django regularmente.

En gran medida este tutorial trata de explicar tanto como es posible para que resulte útil para una amplia audiencia.

Dónde encontrar ayuda:

Si tuvieras algún problema durante este tutorial, por favor posteá un mensaje en django-developers o unite a #django-dev en irc.freenode.net para hablar con otros usuarios de Django que quizás puedan ayudar (ambos en inglés).

Qué cubre este tutorial?

Vamos a recorrer el camino para contribuir un patch a Django por primera vez. Al final de este tutorial, deberías tener un conocimiento básico sobre las herramientas y los procesos involucrados. Específicamente, cubriremos lo siguiente:

- Instalar Git.
- Bajar una copia de desarrollo de Django.
- Correr la test suite de Django.
- Escribir un test para el patch.
- Escribir el código del patch.
- Testear el patch.
- Generar un archivo con los cambios del patch.
- Dónde buscar más información.

Una vez que terminés este tutorial, podés ver el resto de la `documentación sobre contribuir con Django`. Contiene mucha más información y es algo que uno debe leer si quiere convertirse en un contribuidor regular de Django. Si tenés preguntas, probablemente tendrá las respuestas.

1.9.2 Instalar Git

Para este tutorial, necesitaremos tener Git instalado para bajar la versión de desarrollo de Django y generar los archivos con los cambios de nuestro patch.

Para chequear si tenemos Git instalado o no, escribimos `git` en la línea de comandos. Si obtenemos un mensaje diciendo que este comando no se encontró, tendremos que bajar e instalar Git, ver Git's download page.

Si no estás familiarizado con Git, siempre se puede saber más sobre los comandos (una vez instalado) tipeando `git help` en la línea de comandos.

1.9.3 Obtener una copia de la versión de desarrollo de Django

El primer paso para contribuir con Django es obtener una copia del código fuente. En la línea de comandos, usamos el comando `cd` para navegar al directorio donde queremos que viva nuestra copia local de Django.

Bajamos el código fuente de Django usando el siguiente comando:

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

Nota: Para usuarios que quieran usar virtualenv, se puede usar:

```
pip install -e /path/to/your/local/clone/django/
```

(donde `django` es el directorio del repositorio clonado que contiene el archivo `setup.py`) para linkear nuestra copia local en el entorno del virtualenv. Es una gran opción para aislar nuestra copia de desarrollo de Django del resto de nuestro sistema y evitar potenciales conflictos de paquetes.

1.9.4 Volviendo a una revisión anterior de Django

Para este tutorial vamos a usar el ticket #17549 como caso de estudio, así que vamos a volver atrás la historia de versionado de Django en git hasta antes de que el patch se aplicó. Esto nos va a permitir recorrer todos los pasos requeridos al escribir un patch desde cero, incluyendo correr la test suite de Django.

Por más que usemos una revisión más vieja del trunk de Django para los propósitos de este tutorial, siempre se debe usar la revisión actual cuando se trabaja en un patch para un ticket!

Nota: El patch para este ticket fue escrito por Ulrich Petri, y fue aplicado a Django en el commit ac2052ebc84c45709ab5f0f25e685bf656ce79bc. Entonces vamos a usar la revisión anterior a este commit, commit 39f5bc7fc3a4bb43ed8a1358b17fe0521a1a63ac.

Navegamos al directorio raíz de Django (el que contiene `django`, `docs`, `tests`, `AUTHORS`, etc.). Chequeamos la revisión que vamos a usar para el tutorial:

```
git checkout 39f5bc7fc3a4bb43ed8a1358b17fe0521a1a63ac
```

1.9.5 Corriendo la test suite de Django por primera vez

Cuando uno contribuye con Django es muy importante que los cambios no introduzcan bugs en otras áreas de Django. Una manera de chequear que Django todavía funciona después de hacer cambios es correr la test suite. Si todos los tests todavía pasan, entonces uno puede estar razonablemente seguro de que los cambios no han roto Django. Si nunca

corriste la test suite de Django antes, es una buena idea hacerlo una vez antes de comenzar para familiarizarse con la salida.

Podemos correr la test suite simplemente haciendo `cd` al directorio `tests/` de Django y, si usamos GNU/Linux, Mac OS X o algún otro sabor de Unix, ejecutar:

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite
```

Si estamos en Windows, lo de arriba debería funcionar si estamos usando "Git Bash" provisto por la instalación por defecto de Git. GitHub tiene un buen tutorial.

Nota: Si usamos `virtualenv`, podemos omitir `PYTHONPATH=..` cuando corremos los tests. Esto le dice a Python que busque Django en el directorio padre de `tests`. `virtualenv` pone nuestra copia de Django en el `PYTHONPATH` automáticamente.

Ahora nos sentamos y esperamos. La test suite de Django tiene más de 4800 tests, así que puede tomar entre 5 y 15 minutos de correr, dependiendo de la velocidad de nuestra computadora.

Mientras la suite corre, veremos un flujo de caracteres representando el estado de cada test a medida que corre. `E` indica que hubo un error durante el test, y `F` indica que una aserción del test falló. Ambos casos se consideran fallas de test. Por otro lado, `x` y `s` indican fallas esperadas y tests que se saltean, respectivamente. Los puntos indican tests que pasan correctamente.

Los tests que se saltean son típicamente a causa de librerías externas que se requieren para el test pero que no están disponibles; ver *Corriendo todos los tests* para una lista de dependencias y asegurarse de instalarlas para los tests relacionados de acuerdo a los cambios que estemos haciendo (no será necesario para este tutorial).

Una vez que se completan los tests, deberíamos obtener un mensaje que nos informa si la test suite pasó o falló. Como no hemos hecho ningún cambio al código, **debería** haber pasado. Si tuviéramos algún fallo o error, deberíamos asegurarnos que seguimos los pasos previos correctamente. Ver *Corriendo los unit tests* para más información.

Notar que la última revisión del trunk de Django podría no siempre ser estable. Cuando se desarrolla contra trunk, se puede chequear Django's continuous integration builds para determinar si los fallos son específicos de nuestra máquina o también están presentes en los builds oficiales de Django. Si uno cliquea para ver un build particular, puede ver la "Matriz de configuración" que muestra las fallas detalladas por versión de Python y backend de base de datos.

Nota: Para este tutorial y el ticket en el que vamos a trabajar, testear contra SQLite es suficiente, pero es posible (y a veces necesario) *correr los tests usando una base de datos diferente*.

1.9.6 Escribir tests para el ticket

En la mayoría de los casos para que un patch se acepte en Django tiene que incluir tests. Para patches correspondientes a arreglar un bug, esto significa escribir un test de regresión para asegurarse de que el bug no se reintroduce nuevamente. Un test de regresión se debe escribir de tal manera que falle cuando el bug todavía existe y pase cuando el bug se haya arreglado. Para patches de nuevas funcionalidades, es necesario incluir tests que aseguren que la nueva funcionalidad funciona correctamente. También deben fallar cuando la nueva funcionalidad no está presente, y pasar una vez que se haya implementado.

Una buena manera de hacer esto es escribir los tests primero, antes de hacer cualquier cambio al código. Este estilo de desarrollo se llama test-driven development y se puede aplicar tanto a proyectos enteros como a patches. Después de escribir los tests, se corren para estar seguros de que de hecho fallan (ya que no se ha escrito el código que arregla el bug o agrega la funcionalidad todavía). Si los tests no fallan, es necesario arreglarlos para que fallen. Después de todo un test de regresión que pasa independientemente de si el bug está presente o no no es muy útil previniendo que el bug reaparezca más tarde.

Ahora, manos al ejemplo.

Escribir tests para el ticket #17549

El ticket #17549 describe la siguiente funcionalidad a agregar:

> Es útil para un URLField tener la opción de abrir la URL; de otra forma uno podría usar un CharField igualmente.

Para resolver este ticket vamos a agregar un método `render` al `AdminURLFieldWidget` para que se muestre un link cliqueable arriba del widget. Pero antes de hacer ese cambio, vamos a escribir un par de tests que verifiquen que nuestras modificaciones funcionarían correctamente, y lo continuarían haciendo en el futuro.

Navegamos al directorio `tests/regressiontests/admin_widgets/` de Django y abrimos el archivo `tests.py`. Agregamos el siguiente código en la línea 269 justo antes de la clase `AdminFileWidgetTest`:

```
class AdminURLWidgetTest(DjangoTestCase):
    def test_render(self):
        w = widgets.AdminURLFieldWidget()
        self.assertHTMLEqual(
            conditional_escape(w.render('test', '')),
            '<input class="vURLField" name="test" type="text" />'
        )
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example.com')),
            '<p class="url">Currently:<a href="http://example.com">http://example.com</a><br />Change
        )

    def test_render_idn(self):
        w = widgets.AdminURLFieldWidget()
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example-äüö.com')),
            '<p class="url">Currently:<a href="http://xn--example--7za4pnc.com">http://example-äüö.co
        )

    def test_render_quoting(self):
        w = widgets.AdminURLFieldWidget()
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example.com/<sometag>some text</sometag>')),
            '<p class="url">Currently:<a href="http://example.com/%3Csometag%3Esome%20text%3C/sometag
        )
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example-äüö.com/<sometag>some text</sometag>'
            '<p class="url">Currently:<a href="http://xn--example--7za4pnc.com/%3Csometag%3Esome%20te
        )
```

Los nuevos tests chequean que el método `render` que vamos a agregar funciona correctamente en un par de situaciones diferentes.

Pero esto parece un tanto difícil...

Si nunca tuviste que lidiar con tests antes, puede parece un poco difícil escribir los tests a primera vista. Afortunadamente, el testing es un tema *importante* en programación, así que hay mucha información:

- Un buen primer vistazo sobre escribir tests para Django se puede encontrar en la documentación en `Testeando aplicaciones Django`.
- Dive Into Python (libro online gratis para desarrolladores que empiezan con Python) incluye una gran introducción a Unit Testing.

- Después de leer estos, siempre está la documentación de unittest de Python.

Correr los nuevos tests

Recordemos que todavía no hemos hecho ninguna modificación a `AdminURLFieldWidget`, entonces nuestros tests van a fallar. Corramos los tests en el directorio `model_forms_regress` para asegurarnos de que eso realmente sucede. En la línea de comandos, `cd` al directorio `tests/` de Django y corremos:

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite admin_widgets
```

Si los tests corren correctamente, deberíamos ver tres fallas correspondientes a cada uno de los métodos de test que agregamos. Si todos los tests pasan, entonces deberíamos revisar que agregamos los tests en el directorio y clase apropiados.

1.9.7 Escribir el código para el ticket

A continuación vamos a agregar a Django la funcionalidad descripta en el ticket #17549.

Escribir el código para el ticket #17549

Navegamos al directorio `django/django/contrib/admin/` y abrimos el archivo `widgets.py`. Buscamos la clase `AdminURLFieldWidget` en la línea 302 y agregamos el siguiente método `render` después del método `__init__` ya existente:

```
def render(self, name, value, attrs=None):
    html = super(AdminURLFieldWidget, self).render(name, value, attrs)
    if value:
        value = force_text(self._format_value(value))
        final_attrs = {'href': mark_safe(smart_urlquote(value))}
        html = format_html(
            '<p class="url">{0} <a {1}>{2}</a><br />{3} {4}</p>',
            _('Currently:'), flatatt(final_attrs), value,
            _('Change:'), html
        )
    return html
```

Verificar que los tests pasan

Una vez que terminamos las modificaciones, necesitamos verificar que los tests que escribimos anteriormente pasan, para saber si el código que acabamos de escribir funciona correctamente. Para correr los tests en el directorio `admin_widgets`, hacemos `cd` al directorio `tests/` de Django y corremos:

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite admin_widgets
```

Oops, menos mal que escribimos esos tests! Todavía deberíamos ver las tres fallas, con la siguiente excepción:

```
NameError: global name 'smart_urlquote' is not defined
```

Nos olvidamos de agregar el import para ese método. Agregamos el import de `smart_urlquote` al final de la línea 13 de `django/contrib/admin/widgets.py` para que se vea así:

```
from django.utils.html import escape, format_html, format_html_join, smart_urlquote
```

Volvemos a correr los tests y todos deberían pasar. Si no, asegurarse de que se modificó correctamente la clase `AdminURLFieldWidget` como se mostró arriba y que los tests también se copiaron correctamente.

1.9.8 Corriendo la test suite de Django por segunda vez

Una vez que verificamos que nuestro patch y nuestros tests funcionan correctamente, es una buena idea correr la test suite de Django entera para confirmar que nuestro cambio no introdujo ningún bug en otras áreas de Django. Si bien el que la test suite pase no garantiza que nuestro código esté libre de bugs, ayuda a identificar muchos bugs y regresiones que de otra manera podrían pasar desapercibidos.

Para correr la test suite de Django completa, `cd` al directorio `tests/` y corremos:

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite
```

Mientras que no veamos ninguna falla, estamos bien. Notemos que en el fix original también se hace un pequeño cambio de CSS para formatear el widget. Podemos incluir este cambio también, pero lo salteamos por ahora por cuestiones de brevedad.

1.9.9 Escribir documentación

Esta es una nueva funcionalidad, así que debería documentarse. Agregamos lo siguiente desde la línea 925 de `django/docs/ref/models/fields.txt` junto a la documentación existente de `URLField`:

```
.. versionadded:: 1.5

The current value of the field will be displayed as a clickable link above the
input widget.
```

Para mayor información sobre escribir documentación, incluyendo una explicación de qué se trata esto de `versionadded`, se puede consultar en `/internals/contributing/writing-documentation`. Esa página también incluye una explicación de cómo generar una copia de la documentación localmente, para poder prever el HTML que se va a producir.

1.9.10 Generar un patch con los cambios

Ahora es momento de generar un archivo de patch que se pueda subir al Trac o aplicarse a otra copia de Django. Para obtenere un vistazo del contenido de nuestro patch, corremos el siguiente comando:

```
git diff
```

Esto va a mostrar las diferencias entre nuestra copia actual de Django (con los cambios) y la revisión que teníamos inicialmente al principio del tutorial.

Apretamos `q` para volver a la línea de comandos. Si el contenido del patch se veía bien, podemos correr el siguiente comando para guardar el archivo del patch en nuestro directorio actual:

```
git diff > 17549.diff
```

Deberíamos tener un archivo en el directorio raíz de Django llamando `17549.diff`. Este archivo contiene nuestros cambios y debería verse algo así:

```
diff --git a/django/contrib/admin/widgets.py b/django/contrib/admin/widgets.py
index 1e0bc2d..9e43a10 100644
--- a/django/contrib/admin/widgets.py
+++ b/django/contrib/admin/widgets.py
```

```
@@ -10,7 +10,7 @@ from django.contrib.admin.templatetags.admin_static import static
 from django.core.urlresolvers import reverse
 from django.forms.widgets import RadioFieldRenderer
 from django.forms.util import flatatt
-from django.utils.html import escape, format_html, format_html_join
+from django.utils.html import escape, format_html, format_html_join, smart_urlquote
 from django.utils.text import Truncator
 from django.utils.translation import ugettext as _
 from django.utils.safestring import mark_safe
@@ -306,6 +306,18 @@ class AdminURLFieldWidget(forms.TextInput):
             final_attrs.update(attrs)
         super(AdminURLFieldWidget, self).__init__(attrs=final_attrs)

+    def render(self, name, value, attrs=None):
+        html = super(AdminURLFieldWidget, self).render(name, value, attrs)
+        if value:
+            value = force_text(self._format_value(value))
+            final_attrs = {'href': mark_safe(smart_urlquote(value))}
+            html = format_html(
+                '<p class="url">{0} <a {1}>{2}</a><br />{3} {4}</p>',
+                _('Currently:'), flatatt(final_attrs), value,
+                _('Change:'), html
+            )
+        return html
+
 class AdminIntegerFieldWidget(forms.TextInput):
     class_name = 'vIntegerField'

diff --git a/docs/ref/models/fields.txt b/docs/ref/models/fields.txt
index 809d56e..d44f85f 100644
--- a/docs/ref/models/fields.txt
+++ b/docs/ref/models/fields.txt
@@ -922,6 +922,10 @@ Like all :class:`CharField` subclasses, :class:`URLField` takes the optional
 :attr:`~CharField.max_length` argument. If you don't specify
 :attr:`~CharField.max_length`, a default of 200 is used.

+.. versionadded:: 1.5
+
+The current value of the field will be displayed as a clickable link above the
+input widget.

 Relationship fields
 ===================
diff --git a/tests/regressiontests/admin_widgets/tests.py b/tests/regressiontests/admin_widgets/tests
index 4b11543..94acc6d 100644
--- a/tests/regressiontests/admin_widgets/tests.py
+++ b/tests/regressiontests/admin_widgets/tests.py
@@ -265,6 +265,35 @@ class AdminSplitDateTimeWidgetTest(DjangoTestCase):
                     '<p class="datetime">Datum: <input value="01.12.2007" type="text" class="vDateFi
                 )

+class AdminURLWidgetTest(DjangoTestCase):
+    def test_render(self):
+        w = widgets.AdminURLFieldWidget()
```

```
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', '')),
+            '<input class="vURLField" name="test" type="text" />'
+        )
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example.com')),
+            '<p class="url">Currently:<a href="http://example.com">http://example.com</a><br />Chan
+        )
+
+    def test_render_idn(self):
+        w = widgets.AdminURLFieldWidget()
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example-äüö.com')),
+            '<p class="url">Currently:<a href="http://xn--example--7za4pnc.com">http://example-äüö.
+        )
+
+    def test_render_quoting(self):
+        w = widgets.AdminURLFieldWidget()
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example.com/<sometag>some text</sometag>'))
+            '<p class="url">Currently:<a href="http://example.com/%3Csometag%3Esome%20text%3C/somet
+        )
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example-äüö.com/<sometag>some text</sometag
+            '<p class="url">Currently:<a href="http://xn--example--7za4pnc.com/%3Csometag%3Esome%20
+        )
```

```
class AdminFileWidgetTest(DjangoTestCase):
    def test_render(self):
```

1.9.11 Qué hacemos ahora?

Felicitaciones, hemos generado nuestro primer patch para Django! Ahora podemos usar estas habilidades para ayudar a mejorar el código de Django. Generar patches y adjuntarlos a tickets en el Trac es útil, pero, como ahora estamos usando git - se recomiendo adoptar un `workflow orientado a git`.

Como nunca commiteamos nuestros cambios localmente, hacemos lo siguiente para devolver nuestro branch a un buen punto de comienzo (reseteamos nuestros cambios):

```
git reset --hard HEAD
git checkout master
```

Más información para nuevos contribuidores

Antes de ponerse de lleno a escribir patches para Django, hay más información sobre el tema que probablemente deberías leer:

- Deberías asegurarte de leer la documentación de Django sobre `reclamando tickets y enviando patches`. Cubre la etiqueta en Trac, cómo reclamar tickets, el estilo de código esperado para un patch, y

otros detalles importantes.

- Aquellos que contribuyen por primera vez deberían leer también la `documentación para contribuidores por primera vez`. Tiene muchos consejos para quienes son nuevos en esto de ayudar con Django.
- Y si todavía buscás más información sobre contribuir, siempre se puede revisar el resto de la `documentación de Django sobre contribuir`. Contiene mucha información útil y debería ser el primer recurso en busca de respuestas a cualquier pregunta que te pudiera surgir.

Encontrar nuestro primer ticket de verdad

Después de recorrer la información de arriba, estaremos listos para salir a buscar un ticket para el que podamos escribir un patch. Hay que prestar especial atención a los tickets marcados como "easy pickings". Estos tickets suelen ser más simples y son una gran oportunidad para aquellos que quieren contribuir por primera vez. Una vez que estemos familiarizados con el contribuir con Django, podemos pasar a escribir patches para tickets más complicados.

Si queremos empezar ya, se puede pegar una mirada a la lista de tickets simples que necesitan patches y la lista de tickets simples que tienen patches que necesitan mejoras. Si estamos familiarizados con escribir tests, también podemos ver la lista de tickets simples que necesitan tests. Recordar seguir las instrucciones sobre reclamar tickets mencionadas en el link a la documentación en `reclamando tickets y enviando patches`.

Qué sigue?

Después que un ticket tiene un patch, se necesita que un segundo par de ojos lo revisen. Después de subir un patch o enviar un pull request, hay que asegurarse de actualizar la metadata del ticket, seteando los flags del ticket para que diga "has patch", "doesn't need tests", etc. para que otros lo puedan encontrar para revisarlo. Contribuir no necesariamente significa escribir un patch desde cero, revisar patches existentes es también una forma útil de ayudar. Para más detalles, ver `/internals/contributing/triaging-tickets`.

1.10 Acerca de la traducción

Esta es una traducción de la sección de introducción de la documentación de Django, correspondiente a la versión 1.5.

La versión original, en inglés, de este material se puede encontrar en:

https://docs.djangoproject.com/en/1.5/intro/

Por cualquier error, sugerencia o corrección:

https://bitbucket.org/matiasb/django-docs-es/issues

También disponible para bajar en versión PDF.

Ver También:

Si sos nuevo en Python, tal vez quieras empezar por hacerte una idea de cómo es este lenguaje. Django es 100 % Python, entonces si uno tiene un mínimo comfort con Python probablemente puedas obtener mucho más de Django.

Si sos nuevo en lo que a programación se refiere, tal vez te convenga empezar con una leída a esta lista de recursos sobre Python para no-programadores

Si ya conocés otros lenguajes y querés empezar con Python rápidamente, recomendamos Dive Into Python (también disponible en formato papel). Si este libro no sigue tu estilo, hay algunos otros libros sobre Python.

También está disponible el tutorial de Python en castellano!