Django Documentation

Publicación 1.8.x

Django Software Foundation

April 14, 2015

Índice general

Django es un framework web de alto nivel, escrito en Python, que ayuda al desarrollo rápido y a un diseño limpio y pragmático. Construido por desarrolladores experimentados, resuelve una buena parte de los problemas del desarrollo web de tal manera que uno se pueda enfocar en escribir su app sin necesidad de reinventar la rueda. Es gratis y de código abierto.

Sitio web: https://www.djangoproject.com

CAPÍTULO 1

Tutorial

Nuevo en Django? O en desarrollo web en general? Llegaste al lugar indicado: leé este material para ponerte en marcha rápidamente.

1.1 Empezando con Django

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

1.1.1 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.

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 modelo de datos ofrece muchas maneras de representar tus modelos – al día de hoy ha resuelto problemas de esquema de base de datos por años. Aquí un rápido ejemplo: from django.db import models

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

def __str__(self): # __unicode__ on Python 2 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 __str__(self): # __unicode__ on Python 2 return self.headline `mysite/news/models.py`

Instalarlo

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

```
$ python manage.py migrate
```

El comando `migrate` revisa todos los modelos disponibles y crea las tablas en la base de datos para aquellos que todavía no existan, como así también, opcionalmente, proveer un amplio control sobre los esquemas.

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:

```
# Import the models we created from our "news" app
>>> from news.models import Reporter, Article

# No reporters are in the system yet.
>>> Reporter.objects.all()
[]

# Create a new Reporter.
>>> r = Reporter(full_name='John Smith')

# Save the object into the database. You have to call save() explicitly.
>>> r.save()

# Now it has an ID.
>>> r.id
1

# Now the new reporter is in the database.
>>> Reporter.objects.all()
[<Reporter: John Smith>]

# Fields are represented as attributes on the Python object.
>>> r.full_name
'John Smith'

# Django provides a rich database lookup API.
>>> 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.

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

```
# Now the article is in the database.
>>> Article.objects.all()
[<Article: Django is cool>]

# Article objects get API access to related Reporter objects.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# And vice versa: Reporter objects get API access to Article objects.
>>> r.article_set.all()
[<Article: Django is cool>]

# The API follows relationships as far as you need, performing efficient
# JOINs for you behind the scenes.
# This finds all articles by a reporter whose name starts with "John".
>>> Article.objects.filter(reporter__full_name__startswith='John')
[<Article: Django is cool>]

# Change an object by altering its attributes and calling save().
>>> r.full_name = 'Billy Goat'
>>> r.save()

# Delete an object with delete().
>>> r.delete()
```

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: 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) `mysite/news/models.py`

from django.contrib import admin

from . import models

admin.site.register(models.Article) `mysite/news/admin.py`

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.

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 url

from . import views

urlpatterns = [url(r'^articles/([0-9]{4})/$', views.year_archive), url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive), url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),]

`mysite/news/urls.py`

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')`.

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: from django.shortcuts import render

from .models import Article

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

`mysite/news/views.py`

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

Diseñar tus templates

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

Django tiene un path de búsqueda de templates, que permite minimizar la redundancia. En tus settings, especificá una lista de directorios para revisar por templates en `DIRS`. Si un template no existe en el primer directorio, se busca en 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 %} {{ article.headline }} By {{ article.reporter.full_name }} Published {{ article.pub_date|date:"F j, Y" }} { % endfor %} { % endblock %}

`mysite/news/templates/news/year_archive.html`

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", incluyendo el uso de archivos estáticos, podría ser algo como: { % load staticfiles %} <html> <head> <title>{ % block title %}{ % endblock %}</title> </head> <body> { % block content %}{ % endblock %} </body> </html>

`mysite/templates/base.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.

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!

1.1.2 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.

Instalar Python

Siendo un framework web escrito en Python, Django requiere Python. Funciona con Python 2.7, 3.2, 3.3, o 3.4. Estas versiones de Python incluyen una base de datos liviana llamada SQLite, así que no es necesario configurar un motor de base datos inmediatamente.

Para obtener la última versión de Python, ir a https://www.python.org/download/ o instalar a través del administrador de paquetes de tu sistema operativo.

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 3.3.3 (default, Nov 26 2013, 13:33:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Configurar una base de datos

Este paso sólo es necesario si quisieras trabajar con un motor de base de datos más "grande" como PostgreSQL, MySQL u Oracle. Para ello consultá la información sobre la instalación de base de datos.

Borrar versiones anteriores de Django

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

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.

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())
1.8
```

You may have another version of Django installed.

Eso es todo!

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

1.1.3 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())"
```

Si Django está instalado, deberías ver la versión de tu instalación. Si no, obtendrás un error diciendo "No module named django".

Este tutorial está escrito para Django 1.8 y Python 3.2 o mayor. Si la versión de Django no coincide, te podés remitir a la versión del tutorial que corresponda, o actualizar Django a la versión más reciente. Si todavía usás Python 2.7, vas a necesitar ajustar los ejemplos ligeramente como se describe en los comentarios.s

Ver Cómo instalar Django para leer sobre cómo borrar versiones anteriores de Django e instalar una más reciente.

Dónde encontrar ayuda:

Si tenés problemas siguiendo este tutorial, por favor posteá un mensaje a lista de correo django-users o date una vuelta por #django en irc.freenode.net para chatear con otros usuarios de Django que quizás te puedan ayudar.

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 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
```

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. `mysite.urls`).
- `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.

Configurar la base de datos

Ahora editemos `mysite/settings.py`. Es un módulo Python normal, que define variables a nivel módulo que representan los settings de Django.

Por defecto, la configuración usa SQLite. Si sos nuevo en lo que a base de datos se refiere, o solamente te interesa probar Django, está es la opción más simple. SQLite está incluido en Python, entonces no es necesario instalar nada extra. Sin embargo, cuando empieces un proyecto más serio quizás quieras considerar una base de datos más robusta como PostgreSQL, para evitar dolores de cabeza cambiando el motor de base de datos durante el camino.

Si querés usar otro motor de base de datos, instalá los bindings apropiados y cambiá las siguientes claves en DATABASES 'default' para que coincidan con la configuración de tu conexión a la 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 **:setting:'otros backends disponibles <https://docs.djangoproject.com/en/1.8/ref/databases/#third-party-notes>'_**.
- 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.

Si no estás usando SQLite, tenés que agregar parámetros adicionales como USER, PASSWORD, HOST. Para más detalles, ver la documentación de referencia para DATABASES.

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.

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.admin – El sitio de administración. Lo vamos a usar en *la parte 2 de este tutorial*.
- 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.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.

Algunas 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 migrate
```

El comando migrate 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 mysite/settings.py. Verás un mensaje por cada migración que se aplica. Si estás interesado, podés correr el cliente de línea de comandos de tu base de datos y tipear \dt (PostgreSQL), SHOW TABLES; (MySQL), o .schema (SQLite) para ver las tablas que Django creó.

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 migrate. El comando migrate sólo creará las tablas para las apps en INSTALLED_APPS.

El servidor de desarrollo

Verifiquemos que el proyecto Django funciona. Cambiamos al directorio `mysite` de más afuera, si no lo habías hecho, y corremos los siguientes comandos:

```
$ python manage.py runserver
```

Veremos la siguiente salida en la línea de comandos:

```
Performing system checks...

0 errors found
April 14, 2015 - 15:50:53
Django version 1.8, using settings `mysite.settings'
Starting development server 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 buen 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`.

Recarga automática de `runserver`

El servidor de desarrollo recarga automáticamente el código Python en cada request según sea necesario. No es necesario reiniciar el servidor para que los cambios al código tengan efecto. Sin embargo, algunas acciones como agregar archivos no producen un reinicio automático y entonces será necesario reiniciar el servidor a mano en esos casos.

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 que sigue una ciera convención. 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
    admin.py
    migrations/
        __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 **:ref:'principio DRY <https://docs.djangoproject.com/en/1.8/misc/design-philosophies/#dry>'_**. El objetivo es definir el modelo de datos en un lugar y automáticamente derivar lo demás a partir de éste.

Esto incluye las migraciones - a diferencia de Ruby On Rails, por ejemplo, las migraciones son completamente derivadas del archivo de modelos, y son esencialmente una historia que Django puede seguir para actualizar la base de datos y mantenerla en sincronía con tus modelos.

En nuestra simple app poll, vamos a crear dos modelos: `Question` and `Choice`. Una `Question` 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 asociada a una `Question`.

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

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

class Choice(models.Model): question = models.ForeignKey(Question) choice_text = models.CharField(max_length=200) votes = models.IntegerField(default=0)

`polls/models.py`

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_text` 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 `Question.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.

Un `Field` puede tener también varios argumentos opcionales; en este caso, seteamos el valor `default` de `votes` a 0.

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

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 la API Python de acceso a la base de datos para acceder a los objetos `Question` 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 `mysite/settings.py`, y cambiamos el setting `INSTALLED_APPS` para incluir `'polls'`. Se verá algó así: INSTALLED_APPS = ('django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', 'polls',)

`mysite/settings.py`

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

```
$ python manage.py makemigrations polls
```

"Deberíamos ver algo similar a lo siguiente:

```
Migrations for 'polls':
  0001_initial.py:
    - Create model Question
    - Create model Choice
    - Add field question to choice
```

Corriendo `makemigrations`, le estamos diciendo a Django que hemos hecho algunos cambios a nuestros modelos (en este caso, nuevos modelos) y que quisiéramos registrar esos cambios en una *migración*.

Las migraciones es como DJango guarda los cambios a nuestros modelos (y por lo tanto al esquema de base de datos) - son solamente archivos en disco. Podríamos leer la migración de nuestro nuevo modelo si quisiéramos; es el archivo `polls/migrations/0001_initial.py`. No te preocupes, no se espera que uno las lea cada vez que Django crea una nueva, pero están diseñadas para ser editables a mano en caso de que se quiera hacer alguna modificación en la forma que Django aplica los cambios.

Existe un comando que corre las migraciones y administra el esquema de base de datos automáticamente - se llama `migrate`, y llegaremos a él en un momento - pero primero, veamos cuál es el SQL que la migración correría. El comando `sqlmigrate` toma nombres de migraciones y devuelve el SQL respectivo:

```
$ python manage.py sqlmigrate polls 0001
```

Deberías ver algo similar a lo siguiente (reformateado aquí por legibilidad):

```
BEGIN;
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
ALTER TABLE "polls_choice"
  ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
    DEFERRABLE INITIALLY DEFERRED;

COMMIT;
```

Notemos lo siguiente:

- La salida exacta varía de acuerdo a la base de datos que se esté usando. El ejemplo anterior está generado para PostgreSQL.
- Los nombres de las tablas se generan automáticamente combinando el nombre de la app (`polls`) con el nombre, en minúsculas del modelo – `question` 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 un constraint `FOREIGN KEY`. No te preocupes por la parte del `DEFERRABLE`; indica a PostgreSQL no forzar la clave foránea hasta el final de la transacción.
- 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 comando `sqlmigrate` no corre la migración en la base de datos - solamente imprime por pantalla para mostrar cuál es el SQL que Django piensa es requerido. Es útil para chequear lo que Django va a hacer o si uno tiene administradores de base de datos que requieren el SQL para aplicar los cambios.

Si te interesa, también podés correr `python manage.py check`; este comando chequea por cualqueir problema en tu proyecto sin aplicar las migraciones ni tocar la base de datos.

Ahora corramos `migrate` de nuevo para crear las tablas correspondientes a nuestros modelos en la base de datos:

```
$ python manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: staticfiles, messages
  Apply all migrations: admin, contenttypes, polls, auth, sessions
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying <migration name>... OK
```

El comando `migrate` toma todas las migraciones que no se aplicaron (Django lleva registro de cuáles se aplicaron usando una tabla especial en la base de datos llamada `django_migrations`) y las corre contra la base de datos - esencialmente, sincroniza el esquema de la base de datos con los cambios hechos a nuestros modelos.

Las migraciones son muy poderosas y nos permiten cambiar nuestros modelos a lo largo del tiempo, mientras se avanza con el proyecto, sin necesidad de borrar la base de datos o las tablas, y crear nuevas - se especializa en actualizar la base de datos sin perder información. Las veremos en más detalle en una parte más adelante del tutorial, pero por ahora, recordermos los 3 pasos para hacer cambios a nuestros modelos:

- Cambiar nuestros modelos (en `models.py`).
- Correr `python manage.py makemigrations` para crear las migraciones correspondientes a esos cambios
- Correr `python manage.py migrate` para aplicar esos cambios a la base de datos.

La razón por la que hay comandos separados para crear y aplicar migraciones es porque vas a necesitar hacer commit de las migraciones en tu sistema de control de versiones y distribuirlas con tu app; no solamente hacen tu desarrollo más simple, también son reusables por otros desarrolladores y en producción.

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

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 `mysite/settings.py`.

Evitando manage.py

Si preferís no usar `manage.py`, no hay problema. Basta setear la variable de entorno `DJANGO_SETTINGS_MODULE` a `mysite.settings`, levantar un shell de Python, y configurar Django:

```
>>> import django
>>> django.setup()
```

Si esto levanta una excepción `AttributeError`, probablemente estás usando una versión de Django que no coincide con la de este tutorial. Deberías cambiar a la versión del tutorial (o conseguir la versión de Django) correspondiente.

Tenés que correr `python` en el mismo directorio que está `manage.py`, o asegurarte de que ese directorio está en el Python path, para que `import mysite` funcione.

Para más información sobre todo esto, ver la documentación de django-admin.

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

```
>>> from polls.models import Question, Choice   # Import the model classes we just wrote.

# No questions are in the system yet.
>>> Question.objects.all()
[]

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> q.save()

# Now it has an ID. Note that this might say "1L" instead of "1", depending
# on which database you're using. That's no biggie; it just means your
# database backend prefers to return integers as Python long integer
# objects.
>>> q.id
1

# Access model field values via Python attributes.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
[<Question: Question object>]
```

Un minuto. `<Question: Question 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 `__str__()` a `Question` y `Choice`: from django.db import models

class Question(models.Model): # ... def __str__(self): # __unicode__ on Python 2 return self.question_text

class Choice(models.Model): # ... def __str__(self): # __unicode__ on Python 2 return self.choice_text

`polls/models.py`

Es importante agregar el método `__str__()` 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.

`__str__` o `__unicode__`?

En Python 3, es fácil, usamos `__str__()`.

En Python 2, deberíamos en vez definir el método `__unicode__()` que devuelva valores `unicode`. Los modelos de Django tienen una implementación por defecto de `__str__()` que llama a `__unicode__()` y convierte el resultado a un bytestring UTF-8. Esto quiere decir que `unicode(p)` devuelve un string Unicode, y `str(p)` devuelve un bytestring, encodeado en UTF-8. Python hace lo contrario: `object` tiene un método `__unicode__` que llama a `__str__` e interpreta el resultado como un bytestring ASCII. Esta diferencia puede generar confusión.

Si todo esto es mucho ruido, usá Python 3.

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

from django.db import models from django.utils import timezone

class Question(models.Model): # ... def was_published_recently(self): return self.pub_date >= timezone.now() - datetime.timedelta(days=1) `polls/models.py`

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 Question, Choice

# Make sure our __str__() addition worked.
>>> Question.objects.all()
[<Question: What's up?>]

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
[<Question: What's up?>]
>>> Question.objects.filter(question_text__startswith='What')
[<Question: What's up?>]

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
```

```
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
[]

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# Let's delete one of the choices. Use delete() for that.
>>> c = q.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.

1.1.4 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.

Creando un usuario admin

Primero vamos a necesitar crear un usuario que pueda loguearse al sitio de administración. Corremos el siguiente comando:

```
$ python manage.py createsuperuser
```

Ingresamos el nombre usuario que querramos y presionamos enter.

```
Username: admin
```

Deberemos ingresar la dirección de email:

```
Email address: admin@example.com
```

El último paso es ingresar el password. Deberemos ingresarlo dos veces, la segunda como una confirmación de la primera.

```
Password: **********
Password (again): *********
Superuser created successfully.
```

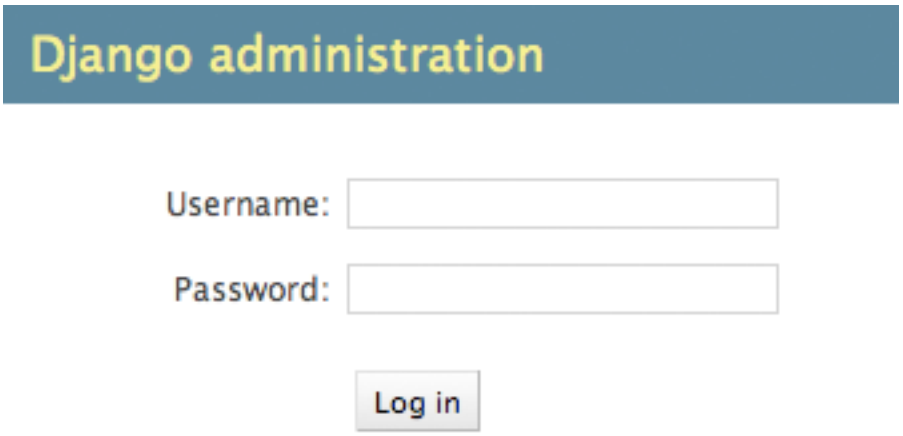
Levantar el servidor de desarrollo

El sitio de admin de Django está habilitado por defecto. Levantemos el servidor de desarrollo y exploremoslo.

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:



Dado que translation está activado por defecto, la pantalla de login podría mostrarse en tu propio idioma, dependiendo de la configuración de tu browser y de si Django tiene las traducciones para dicho idioma.

No coincide con lo que ves?

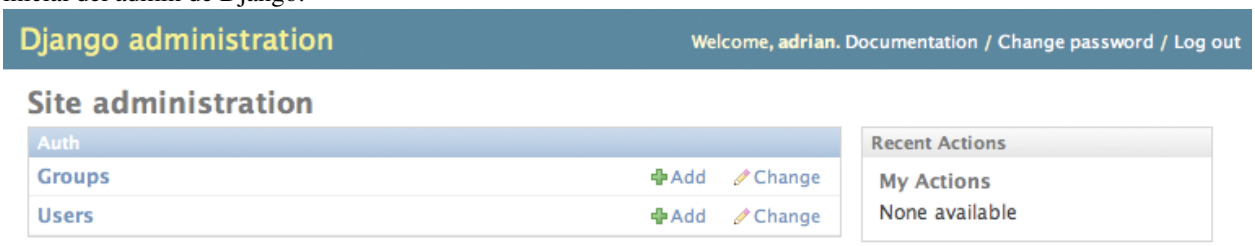
Si en este punto en lugar de la pantalla de login de arriba obtenés una página de error reportando algo como:

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

entonces probablemente estás usando una versión de Django que no coincide con la de este tutorial. Deberías cambiar a una versión anterior del tutorial o a una versión más nueva de Django.

Ingresar al admin

Ahora intentemos loguearnos con la cuenta de superusuario que creamos en el paso anterior. Deberíamos ver la página inicial del admin de Django:



Deberíamos ver unos pocos tipos de contenido editable: grupos y usuarios. Éstos vienen provistos por `django.contrib.auth`, el framework de autenticación que viene con Django.

Hacer la app poll modificable en el admin

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

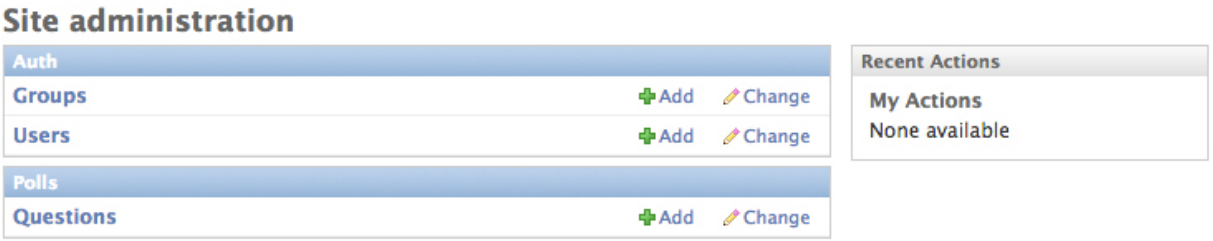
Hay una cosa que hacer: necesitamos decirle al admin que los objetos `Question` tengan una interfaz de administración. Para esto abrimos el archivo `polls/admin.py`, y lo editamos para que tenga el siguiente contenido: from django.contrib import admin

from .models import Question

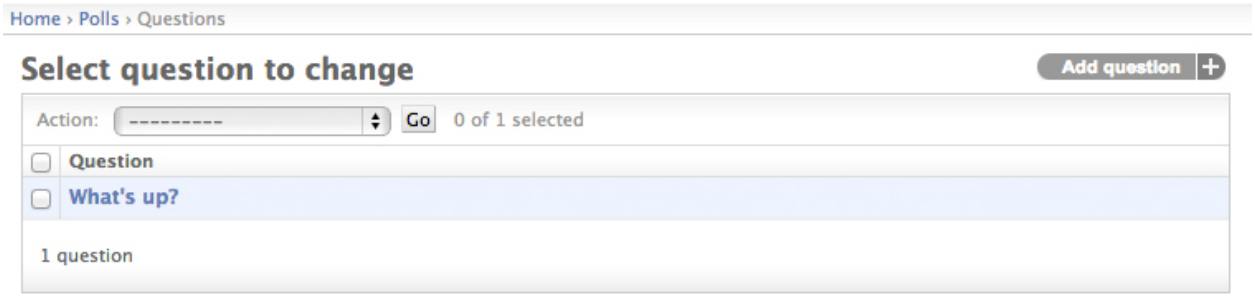
admin.site.register(Question) `polls/admin.py`

Explorar la funcionalidad del admin

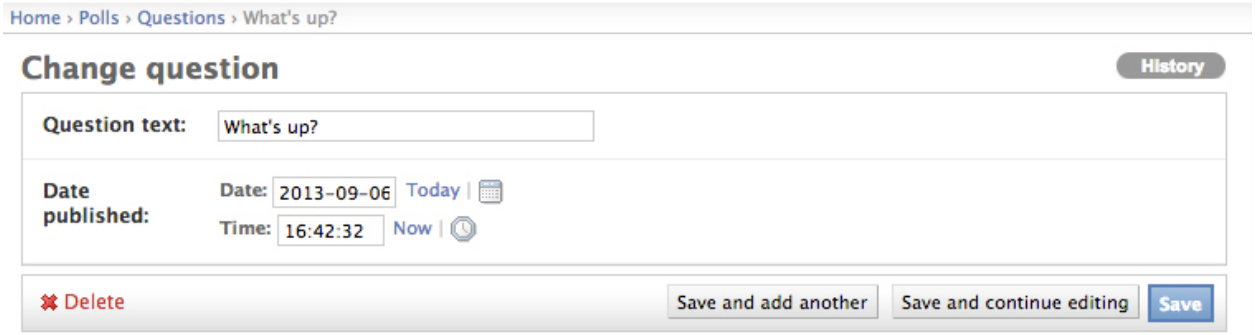
Ahora que registramos `Question`, Django sabe que se debe mostrar en la página del admin:



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



Cliqueamos en ella para editarla:



Cosas para notar aquí:

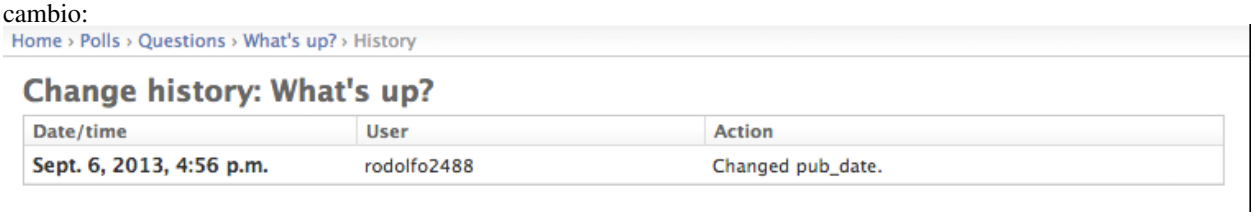
- El form es generado automáticamente a partir del modelo `Question`.
- Los diferentes tipos de campo del modelo (`DateTimeField`, `CharField`) se corresponden con el widget HTML apropiado. Cada tipo de campo sabe cómo mostrarse en el admin de Django.
- Cada `DateTimeField` tiene atajos JavaScript. La 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:

- Guardar (Save) – Guarda los cambios y nos devuelve a la página listado para este tipo de objeto.
- Grabar y continuar editando (Save and continue editing) – Guarda los cambios y recarga la página del admin de este objeto.
- Grabar y agregar otro (Save and add another) – Guarda los cambios y carga un form nuevo, en blanco, que permite agregar un nuevo objeto de este tipo.
- Eliminar (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:



Personalizar el form del admin

Lleva unos pocos minutos maravillarse por todo el código que no tuvimos que escribir. Sólo registrando el modelo `Question` con `admin.site.register(Question)`, 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(Question)` por: from django.contrib import admin

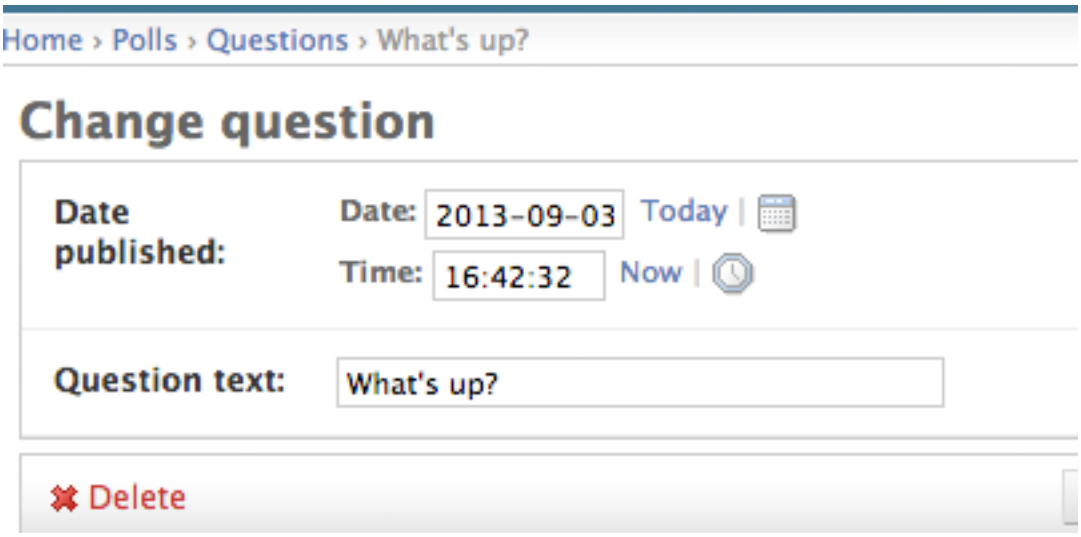
from .models import Question

class QuestionAdmin(admin.ModelAdmin): fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin) `polls/admin.py`

Seguiremos este patrón – creamos un objeto model admin, 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.

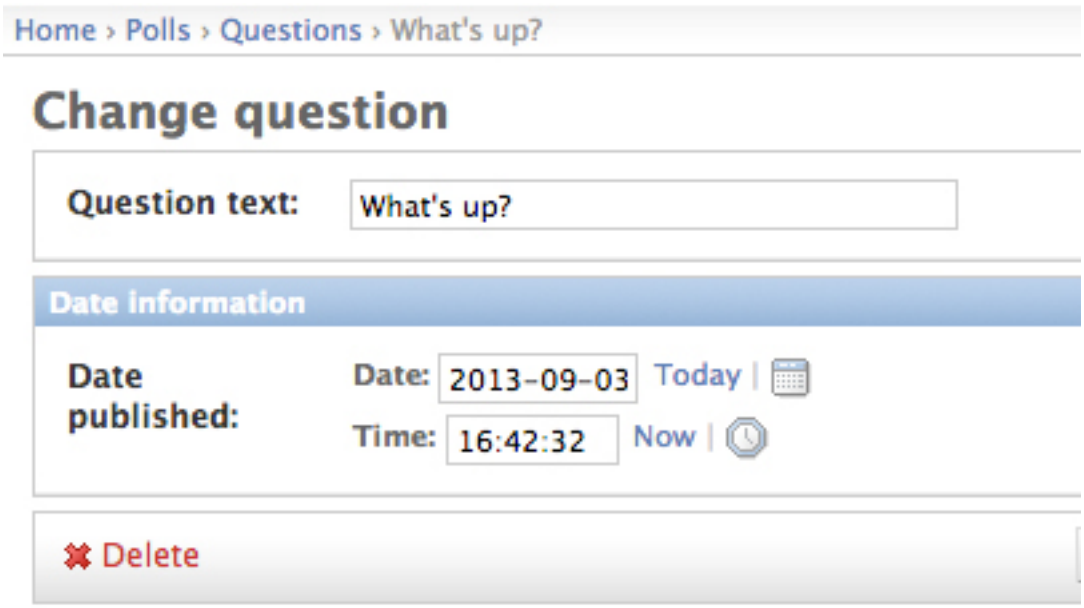
Y hablando de forms con docenas de campos, podríamos querer dividir el form en fieldsets: from django.contrib import admin

from .models import Question

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

admin.site.register(Question, QuestionAdmin) `polls/admin.py`

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



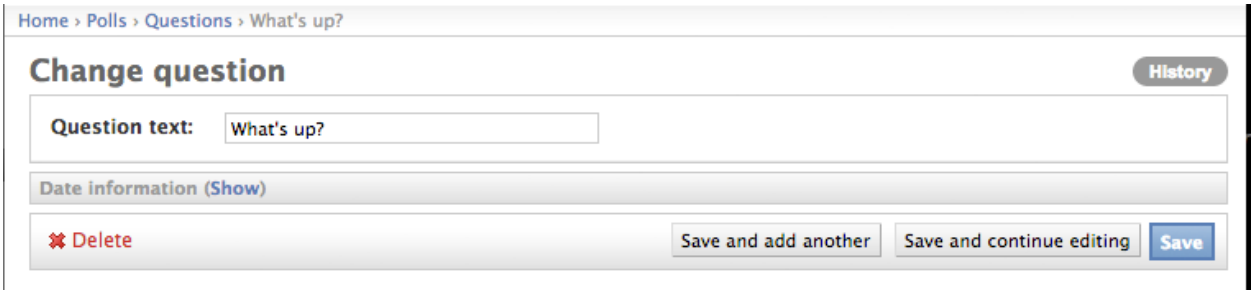
Podemos asignar clases HTML arbitrarias a cada fieldset. Django provee una clase `"collapse"` que muestra el fieldset inicialmente plegado. Esto es útil cuando uno tiene un form largo que contiene ciertos campos que no se usan

normalmente: from django.contrib import admin

from .models import Question

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

admin.site.register(Question, QuestionAdmin) `polls/admin.py`



Agregando objetos relacionados

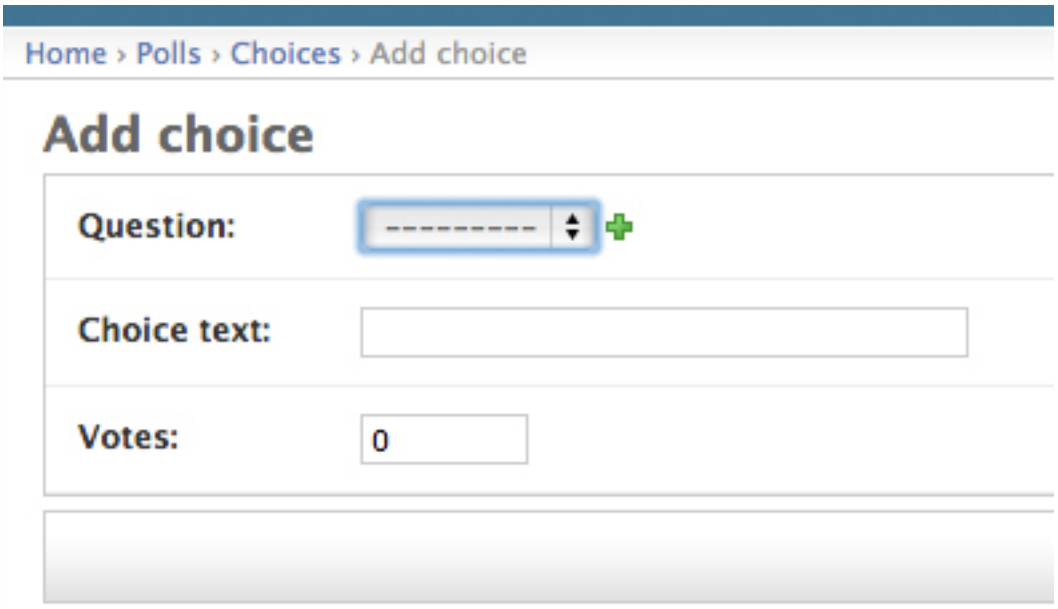
OK, tenemos nuestra página de admin para el modelo Question. Pero un objeto `Question` tiene múltiple `Choices`, y la página de admin no nos muestra estas opciones.

Todavía.

Hay dos formas de resolver este problema. La primera es registrar `Choice` con el admin como hicimos con `Question`. Esto es fácil: from django.contrib import admin

from .models import Choice, Question # ... admin.site.register(Choice) `polls/admin.py`

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



En ese form, el campo "Question" 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 "Question". 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 question". Si uno agrega una pregunta mediante este form y hace click en "Save", Django va a guardar la pregunta 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 `Question`. Hagamos que eso ocurra.

Borramos la llamada a `register()` para el modelo `Choice`. Editamos el código que registra `Question` para que se vea así: from django.contrib import admin

from .models import Choice, Question

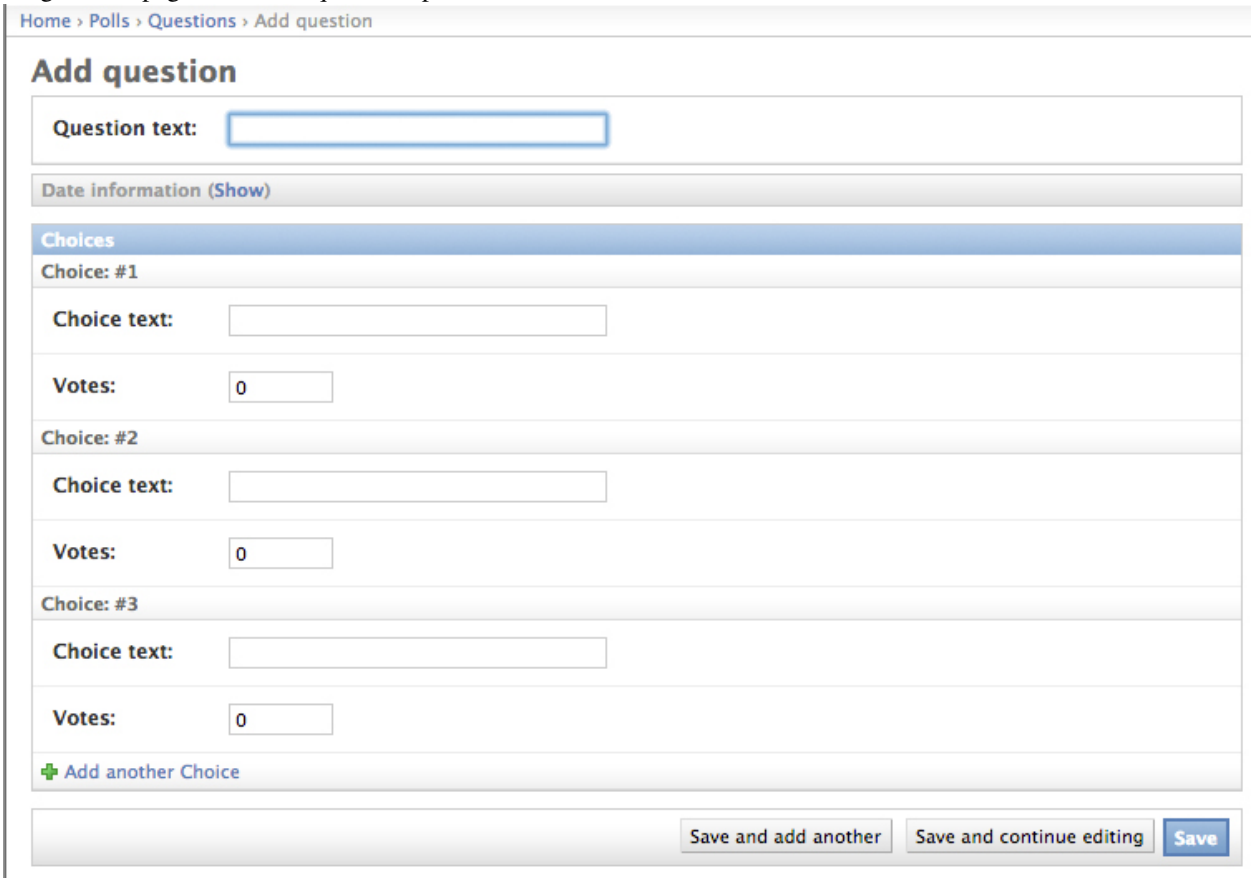
class ChoiceInline(admin.StackedInline): model = Choice extra = 3

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

admin.site.register(Question, QuestionAdmin) `polls/admin.py`

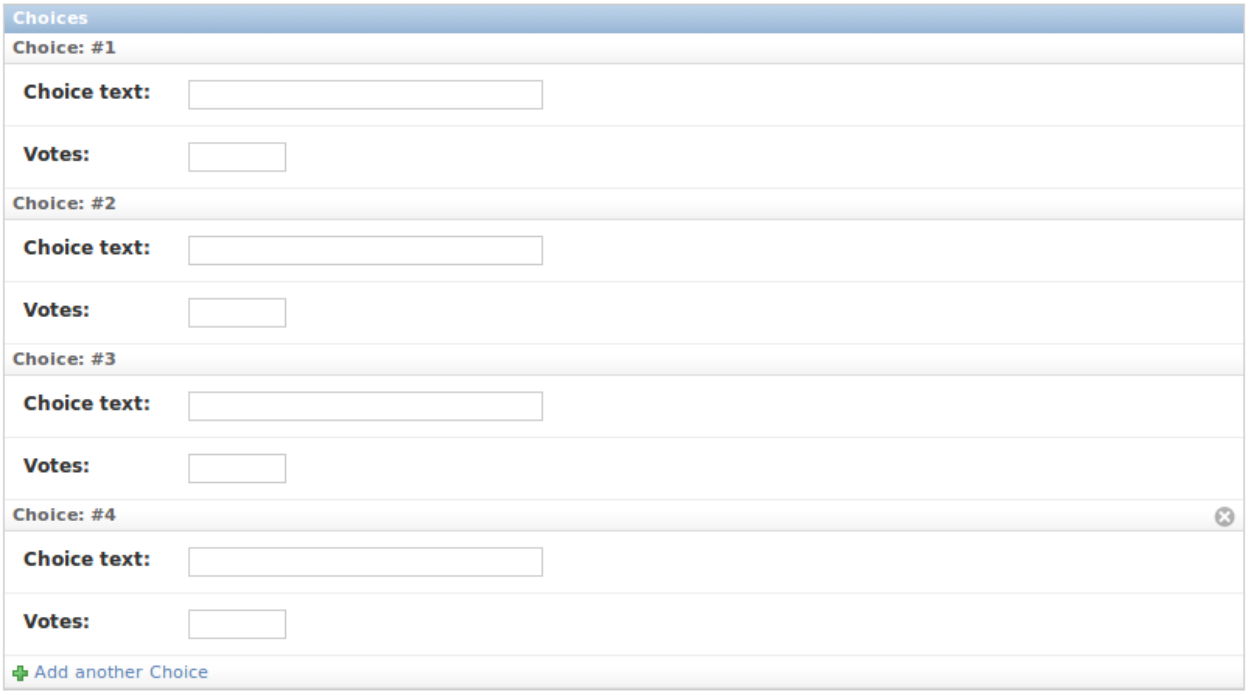
Esto le dice a Django: "los objetos `Choice` se editan en la página de admin de `Question`. Por defecto, proveer los campos para 3 choices".

Cargamos la página de "Add question" para ver cómo se ve:



Funciona así: hay 3 slots para Choices relacionados – como especifica `extra` – y cada vez que uno vuelve a la página de "Change" para un objeto ya creado, se muestran otros 3 slots extra.

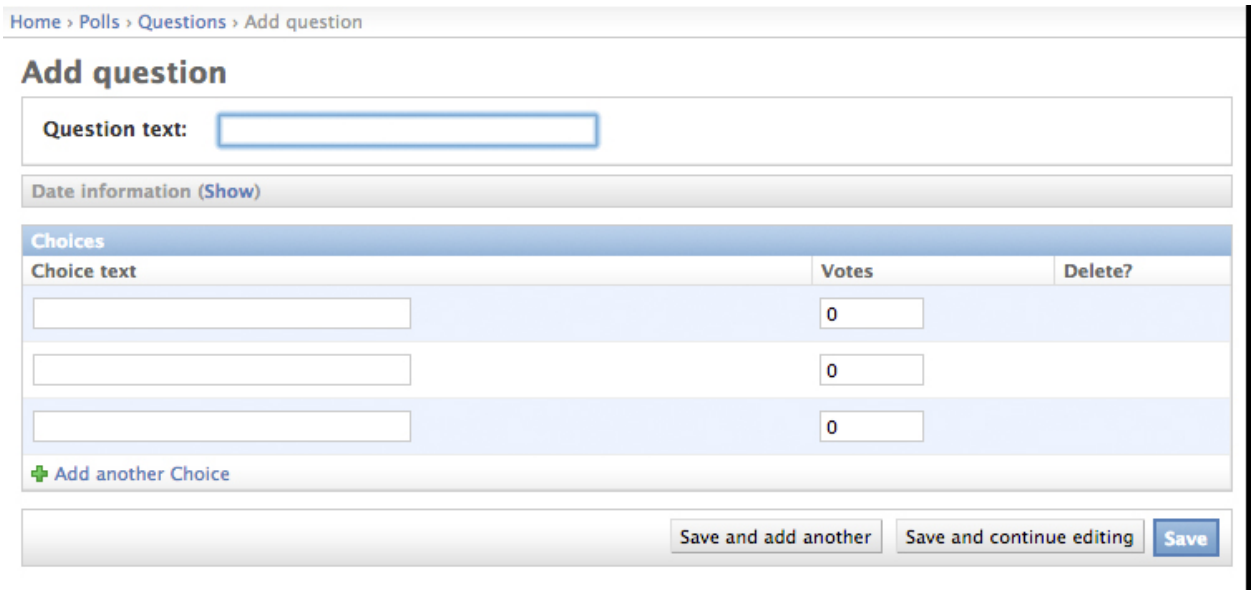
Al final de los 3 slots actuales se encuentra un link para agregar una nueva Choice ("Add another Choice"). Si se hace click en el mismo, se agrega un nuevo slot. Si uno quisiera borar uno de los slots agregados, se puede hacer click en la X en la esquina superior derecha del slot. Notar que no se puede borrar de esta manera los slots originales. Esta imagen muestra un slot agregado como se describió:



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): #...

`polls/admin.py`

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

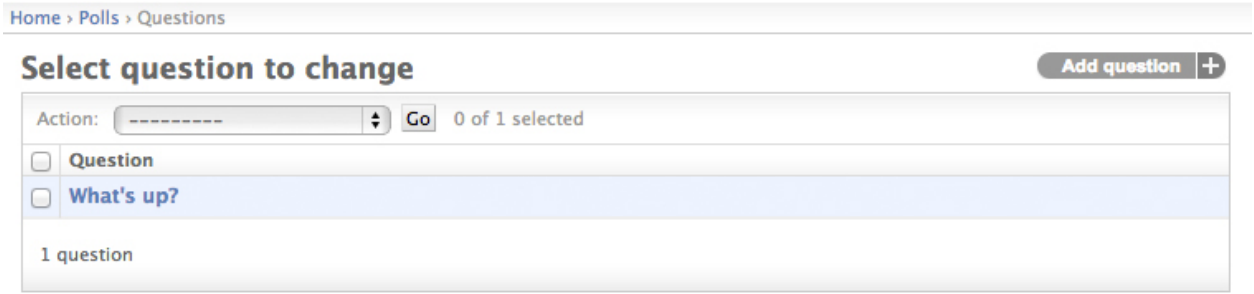


Notemos que hay una columna extra con la opción "Delete?" que nos permite borrar filas que hayamos agregado usando el botón "Add Another Choice" y filas que ya hayan sido guardadas.

Personalizar la página listado

Ahora que la página de admin de Question se ve bien, hagamos algunos tweaks a la página de listado – la que lista todas las preguntas 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 QuestionAdmin(admin.ModelAdmin): # ... list_display = ('question_text', 'pub_date') `polls/admin.py`

Sólo por si acaso, incluyamos también el método `was_published_recently` que definimos en la primera parte: class QuestionAdmin(admin.ModelAdmin): # ... list_display = ('question_text', 'pub_date', 'was_published_recently') `polls/admin.py`

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 `polls/models.py`) como sigue: class Question(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?'

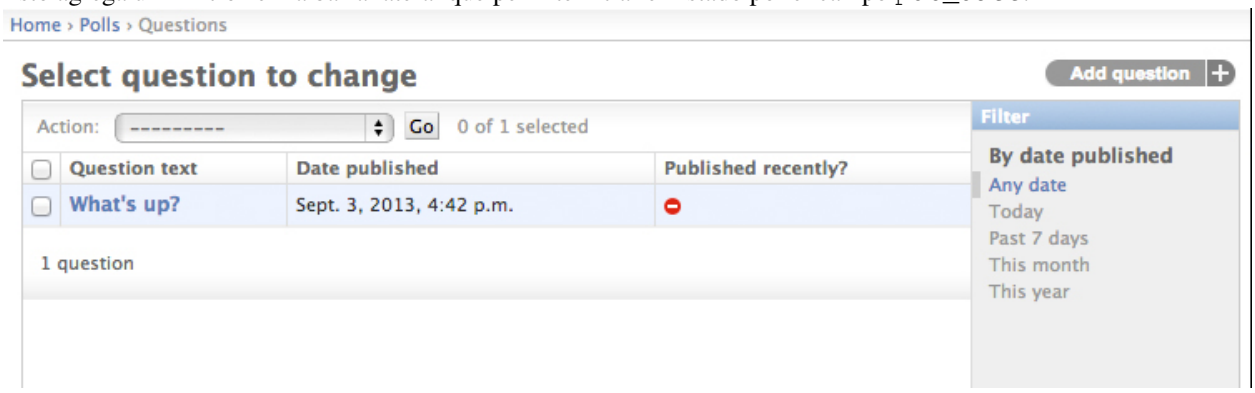
`polls/models.py`

Para más información sobre estas propiedades, ver `list_display`.

Editamos el archivo `polls/admin.py` de nuevo y agregamos una mejora a la página de listado de preguntas: filtros usando `list_filter`. Agregamos la siguiente línea a `QuestionAdmin`:

```
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_text']
```

Esto agrega un campo de búsqueda al tope del listado. Cuando alguien ingresa términos de búsqueda, Django va a

buscar sobre el campo `question_text`. 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 hacérselo simple a la base de datos.

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 desde columnas` funcionan como uno esperaría.

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.

Personalizando los templates de tu *proyecto*

Creamos un directorio `templates` en el directorio del proyecto (el que contiene `manage.py`). 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.

Abrimos el archivo de settings (recordemos, `mysite/settings.py`) y agregamos la opción `DIRS` en el setting `TEMPLATES`: TEMPLATES = [{ 'BACKEND': 'django.template.backends.django.DjangoTemplates', 'DIRS': [os.path.join(BASE_DIR, 'templates')], 'APP_DIRS': True, 'OPTIONS': { 'context_processors': ['django.template.context_processors.debug', 'django.template.context_processors.request', 'django.contrib.auth.context_processors.auth', 'django.contrib.messages.context_processors.messages',], }, },]

mysite/settings.py

`DIRS` es una lista de directorios en los que Django chequea para cargar templates; es un path de búsqueda.

Ahora creamos un directorio llamado `admin` dentro de `templates`, y copiamos el template `admin/base_site.html` del directorio de templates del admin de Django por defecto, del código fuente de Django mismo (`django/contrib/admin/templates`), a ese directorio.

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 `{{ site_header|default:_('Django administration') }}` (incluyendo las llaves) por el nombre de nuestro sitio. Deberías terminar con una sección de código como:

```
{% block branding %}
<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>
{% endblock %}
```

Usamos esta metodología para enseñarte la manera en que se personalizan templates. En un proyecto real, probablemente usaríamos el atributo `django.contrib.admin.AdminSite.site_header` para cambiar este valor en particular.

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 y aplicar los cambios.

Personalizando los templates de tu *aplicación*

Los lectores astutos se estarán preguntando: si `DIRS` estaba vacío por defecto, cómo es que Django encuentra los templates del admin? La respuesta es que, como `APP_DIRS` está seteado en `True`, Django automáticamente busca un subdirectorio `templates/` en cada paquete de aplicación, para usar como fallback (no olvidarse que `django.contrib.admin` es una aplicación).

Nuestra aplicación no es muy compleja y no necesita personalizar templates del admin. Pero si creciera de forma más sofisticada y requiriera cambios a los templates base del admin para alguna funcionalidad, sería más razonable modificar los templates a nivel *aplicación* en lugar de *proyecto*. De esta manera, se podría incluir la aplicación `polls` en cualquier nuevo proyecto y asegurarse de que los templates personalizados estuvieran disponibles.

Ver la documentación de carga de templates para más información sobre cómo Django busca los templates.

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 nuestra aplicación.

1.1.5 Escribiendo tu primera Django app, parte 3

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

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 preguntas, vamos a tener las siguientes cuatro views:

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

En Django, las páginas web y otro contenido se manejan mediante views. Cada view se representa por una simple función Python (o método, en el caso de views basadas en clases). Django elige una view examinando la URL que se pidió (para ser precisos, la parte de la URL después del nombre de dominio).

En la web puede que te hayas encontrado con cosas como "ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967
Estarás encantado de saber que Django permite definir *patrones de URL* mucho más elegantes.

Un patrón de URL es simplemente la forma genérica de una URL - por ejemplo: `/newsarchive/<year>/<month>/`.

Para llegar de una URL a una view, Django usa lo que se conoce como 'URLconfs'. Un URLconf mapea patrones de URL (descriptos como expresiones regulares) a views.

Este tutorial provee las instrucciones básicas para el uso de URLconfs, para más información referirse a `django.core.urlresolvers`.

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 polls index.") `polls/views.py`

Esta es la view más simple posible en Django. Para llamar a esta view, necesitamos mapearla a una URL - y para ello necesitamos un URLconf.

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 url

from . import views

urlpatterns = [url(r'^$', views.index, name='index'),] `polls/urls.py`

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 include, url from django.contrib import admin

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

`mysite/urls.py`

No coincide con lo que ves?

Si estás viendo `admin.autodiscover()` antes de la definición de `urlpatterns`, probablemente estás usando una versión de Django que no coincide con la de este tutorial. Deberías cambiar a una versión anterior del tutorial o a una versión más nueva de Django.

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 polls 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 URLconf 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 por su nombre).

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, question_id): return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id): response = "You're looking at the results of question %s." return HttpResponse(response % question_id)

def vote(request, question_id): return HttpResponse("You're voting on question %s." % question_id)
`polls/views.py`

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

from . import views

urlpatterns = [# ex: /polls/ url(r'^$', views.index, name='index'), # ex: /polls/5/ url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'), # ex: /polls/5/results/ url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'), # ex: /polls/5/vote/ url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),]
`polls/urls.py`

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<question_id>[0-9]+)/$'`, resultando en una llamada a la view `detail()` de la forma:

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

La parte `question_id='34'` surge de `(?P<question_id>[0-9]+)`. 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<question_id>` define el nombre que se usará para identificar la coincidencia; y `[0-9]+` 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:

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

Pero no hagan esto. No tiene sentido.

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 preguntas más recientes en el sistema, separadas por comas, de acuerdo a la fecha de publicación: from django.http import HttpResponse

from .models import Question

def index(request): latest_question_list = Question.objects.order_by('-pub_date')[:5] output = ', '.join([p.question_text for p in latest_question_list]) return HttpResponse(output)

Leave the rest of the views (detail, results, vote) unchanged `polls/views.py`

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, creando un template que la view pueda usar.

Primero, creamos un directorio llamado `templates` en nuestro directorio `polls`. Django va a buscar los templates allí.

El setting `TEMPLATES` de nuestro proyecto describe cómo Django va a cargar y renderizar templates. Por defecto se configura como backend `DjangoTemplates`, con la opción `APP_DIRS` en True. Por convención `DjangoTemplates` busca por un subdirectorio "templates" en cada una de las aplicaciones en `INSTALLED_APPS`. Esta es la manera en que Django sabe encontrar los templates de nuestra aplicación polls sin tener que modificar la opción `DIRS`, como hicimos en *Tutorial 2*.

Organizando los templates

Podríamos tener todos nuestros templates juntos, en un gran directorio templates y funcionaría perfectamente bien. Sin embargo, este template pertenece a la aplicación polls entonces, a diferencia del template del admin que creamos en la parte anterior, vamos a poner este en el directorio de templates de la aplicación (`polls/templates`) en lugar de aquel del proyecto (`templates`). Discutiremos más en detalle en *el tutorial de apps reusables* el *por qué* de esto.

En el directorio `templates` que acabamos de crear, creamos otro directorio llamado `polls`, y allí creamos un archivo `index.html`. En otras palabras, nuestro template debería estar en `polls/templates/polls/index.html`. Como la carga de templates basada en `app_directories` funciona como se describió arriba, nos podemos referir a este template en Django simplemente como `polls/index.html`.

Espacio de nombre en templates

Ahora *podríamos* continuar poniendo nuestros templates directamente en `polls/templates` (en lugar de crear otro subdirectorio `polls`), pero no sería una buena idea. Django elegirá el primer template que encuentre tal que el nombre coincida, y si tenemos un template con el mismo nombre en una aplicación *diferente*, Django no podrá distinguir entre ellos. Necesitamos apuntar a Django al correcto, y la forma más fácil de asegurarse de esto es usando *espacio de nombres* para ellos. Esto es, poner los templates dentro de *otro* directorio llamado como la aplicación correspondiente.

Pongamos el siguiente código en ese template: { % if latest_question_list %} { % for question in latest_question_list %} {{ question.question_text }} { % endfor %} { % else %} No polls are available. { % endif %}

`polls/templates/polls/index.html`

Ahora actualicemos nuestra view `index` en `polls/views.py` para usar el template: from django.http import HttpResponse from django.template import RequestContext, loader

from .models import Question

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

`polls/views.py`

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

Carguemos la página apuntando el browser a "/polls/", y deberíamos ver una lista conteniendo la pregunta "What's up" del Tutorial 1. El link apunta a la página de detall de la pregunta.

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 .models import Question

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

`polls/views.py`

Una vez que hayamos hecho esto para todas estas views, ya no necesitamos importar `loader`, `RequestContext` 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` con el template renderizado con el contexto dado.

Levantando un error 404

Ahora veamos la view de detalle de una pregunta – la página que muestra el texto de la pregunta. Aquí está la view: from django.http import Http404 from django.shortcuts import render

from .models import Question # ... def detail(request, question_id): try: question = Question.objects.get(pk=question_id) except Question.DoesNotExist: raise Http404("Question does not exist") return render(request, 'polls/detail.html', {'question': question})
polls/views.py

El nuevo concepto aquí: la view levanta una excepción `Http404` si no existe una pregunta con el 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: {{ question }}
polls/templates/polls/detail.html

para ponernos en marcha por ahora.

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 get_object_or_404, render

from .models import Question # ... def detail(request, question_id): question = get_object_or_404(Question, pk=question_id) return render(request, 'polls/detail.html', {'question': question})
polls/views.py

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.

Usando el sistema de templates

Volvamos a la view `detail()` de nuestra aplicación. Dada la variable de contexto `question`, veamos como podría lucir el template `polls/detail.html`: <h1>{{ question.question_text }}</h1> {% for choice in question.choice_set.all %} {{ choice.choice_text }} {% endfor %}
polls/templates/polls/detail.html

El sistema de templates usa sintaxis de punto para acceder a los atributos de variable. En el ejemplo de `{{ question.question_text }}`, Django primero hace una búsqueda de diccionario sobre el objeto `question`. 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 %}`: `question.choice_set.all` se interpreta como el código Python `question.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.

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/{{ question.id }}/">{{ question.question_text }}</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 con 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' question.id %}">{{ question.question_text }}</a></li>
```

La forma en que esto funciona es buscando la definición de la URL especificada en el módulo `polls.urls`. Podemos ver dónde se define el nombre 'detail' de la URL aquí:

```
...
# the 'name' value as called by the {% url %} template tag
url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
...
```

Si uno quisiera cambiar la URL de la view de detalle de pregunta 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<question_id>[0-9]+)/$', views.detail, name='detail'),
...
```

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 al resolver 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 include, url from django.contrib import admin

urlpatterns = [url(r'^polls/', include('polls.urls', namespace="polls")), url(r'^admin/', include(admin.site.urls)),]
`mysite/urls.py`

Ahora cambiamos el template `polls/index.html` de: {{ question.question_text }} `polls/templates/polls/index.html`

a que apunte a la view de detalle con el espacio de nombres: {{ question.question_text }}
polls/templates/polls/index.html

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.

1.1.6 Escribiendo tu primera Django app, parte 4

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

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>{{ question.question_text }}</h1>

{ % if error_message %} {{ error_message }} { % endif %}

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

Un repaso rápido:

- El template de arriba muestra un radio button para cada opción de la pregunta. El `value` de cada radio es el ID asociado a cada opción de la pregunta. El `name`, es `"choice"`. Esto significa que cuando alguien elige una de las opciones y envía el form, se envía `choice=#` como data del POST, donde # es el ID de la opción elegida. Este es un concepto básico de forms HTML.
- Establecemos como `action` del form `{ % url 'polls:vote' question.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<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
polls/urls.py

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 .models import Choice, Question # ... def vote(request, question_id): p = get_object_or_404(Question, pk=question_id) try: selected_choice = p.choice_set.get(pk=request.POST['choice']) except (KeyError, Choice.DoesNotExist): # Redisplay the question voting form. return render(request, 'polls/detail.html', { 'question': p, 'error_message': "You didn't select a choice.", }) else: selected_choice.votes += 1 selected_choice.save() # Always return an HttpResponseRedirect after successfully dealing # with POST data. This prevents data from being posted twice if a # user hits the Back button. return HttpResponseRedirect(reverse('polls:results', args=(p.id,)))
`polls/views.py`

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 pregunta. Escribamos esta view: from django.shortcuts import get_object_or_404, render

def results(request, question_id): question = get_object_or_404(Question, pk=question_id) return render(request, 'polls/results.html', {'question': question})
`polls/views.py`

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>{{ question.question_text }}</h1>

 { % for choice in question.choice_set.all %} {{ choice.choice_text }} – {{ choice.votes }} vote{{ choice.votes|pluralize }} { % endfor %}

Vote again? `polls/templates/polls/results.html`

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.

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 preguntas, 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" (views genéricas).

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.

Arreglando URLconf

Primero, abrimos el URLconf `polls/urls.py` y lo cambiamos de la siguiente manera: from django.conf.urls import url

from . import views

urlpatterns = [url(r'^$', views.IndexView.as_view(), name='index'), url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'), url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as_view(), name='results'), url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),]
`polls/urls.py`

Notar que el nombre del patrón que se busca en las segunda y tercera expresiones regulares cambió de `<question_id>` a `<pk>`.

Arreglando views

A continuación vamos a borrar nuestras viejas views `index`, `detail`, y `results` para usar las generic views de Django en vez. Para ello, abrimos el archivo `polls/views.py` y cambiamos: from django.shortcuts import get_object_or_404, render from django.http import HttpResponseRedirect from django.core.urlresolvers import reverse from django.views import generic

from .models import Choice, Question

class IndexView(generic.ListView): template_name = 'polls/index.html' context_object_name = 'latest_question_list'

def get_queryset(self): """Return the last five published questions.""" return Question.objects.order_by('-pub_date')[:5]

class DetailView(generic.DetailView): model = Question template_name = 'polls/detail.html'

class ResultsView(generic.DetailView): model = Question template_name = 'polls/results.html'

def vote(request, question_id): ... # same as above `polls/views.py`

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 atributo `model`.
- La view genérica `DetailView` espera el valor de clave primaria capturado de la URL con nombre `\"pk\"`, entonces cambiamos `question_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/question_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 `question` y `latest_question_list`. Para `DetailView` la variable `question` es provista automáticamente – como estamos usando un modelo Django (`Question`), Django puede determinar un nombre adecuado para la variable de contexto. Sin embargo, para ListView, el nombre de variable de contexto generado automáticamente es `question_list`. Para sobreescribir este valor, pasamos la opción `context_object_name`, especificando que queremos usar `latest_question_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.

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 preguntas.

1.1.7 Escribiendo tu primera Django app, parte 5

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

Introduciendo testing automatizado

Qué son los tests automatizados?

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

El testing opera a 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!

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.

Escribiendo nuestro primer test

Identificamos un bug

Por suerte, hay un pequeño bug en la aplicación `polls` que podemos arreglar: el método `Question.was_published_recently()` devuelve `True` si una `Question` 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 pregunta cuya fecha es en el futuro; veremos que el listado de preguntas nos dice que fue publicada recientemente.

También podemos verlo usando el `shell`:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.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 lugar por convención para los tests de una aplicación es el archivo `tests.py` - el corredor de tests va a buscar los tests allí automáticamente, en aquellos archivos cuyo nombre empiece po `test`.

Ponemos el siguiente código en el archivo `tests.py` en la aplicación `polls`: import datetime

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

from .models import Question

class QuestionMethodTests(TestCase):

def test_was_published_recently_with_future_question(self): “”” was_published_recently() should return False for questions whose pub_date is in the future. “”” time = timezone.now() + datetime.timedelta(days=30) future_question = Question(pub_date=time) self.assertEqual(future_question.was_published_recently(), False)

`polls/tests.py`

Lo que hemos hecho aquí es crear una subclase de `django.test.TestCase` con un método que crea una instancia de `Question` 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_question (polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_question
    self.assertEqual(future_question.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_question` creó una instancia de `Question` 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: `Question.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
`polls/models.py`

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 reintroducir 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_question(self): “”” was_published_recently() should return False for questions whose pub_date is older than 1 day. “”” time = timezone.now() - datetime.timedelta(days=30) old_question = Question(pub_date=time) self.assertEqual(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self): “”” was_published_recently() should return True for questions whose pub_date is within the last day. “”” time = timezone.now() - datetime.timedelta(hours=1) recent_question = Question(pub_date=time) self.assertEqual(recent_question.was_published_recently(), True)
`polls/tests.py`

Y ahora tenemos tres tests que confirman que `Question.was_published_recently()` devuelve valores sensatos para preguntas 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.

Testeando una view

La aplicación polls no discrimina: va a publicar cualquier pregunta, 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 pregunta 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()
```

`setup_test_environment()` instala un renderizador de templates que nos permitirá examinar algunos atributos adicionales en las respuestas, tales como `response.context`, que de otra forma no estarían disponibles. Notar que este método *no configura* una base de datos para testing, entonces lo que corramos a continuación va a ser contra la base de datos existente y la salida podría diferir dependiendo de las preguntas que hayamos creado.

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 import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

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

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded 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'
>>> # note - you might get unexpected results if your ``TIME_ZONE``
>>> # in ``settings.py`` is not correct. If you need to change it,
>>> # you will also need to restart your shell session
>>> from polls.models import Question
>>> from django.utils import timezone
>>> # create a Question and save it
>>> q = Question(question_text="Who is your favorite Beatle?", pub_date=timezone.now())
>>> q.save()
>>> # check the response once again
>>> 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
>>> # If the following doesn't work, you probably omitted the call to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
[<Question: Who is your favorite Beatle?>]
```

Mejorando nuestra view

La lista de preguntas 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`: class IndexView(generic.ListView): template_name = 'polls/index.html' context_object_name = 'latest_question_list'

def get_queryset(self): """Return the last five published questions."""" return Question.objects.order_by('-pub_date')[:5] `polls/views.py`

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

Necesitamos corregir el método `get_queryset` y cambiarlo de tal manera que también chequee la fecha, comparándola con `timezone.now()`. Primero necesitamos hacer un import: from django.utils import timezone
`polls/views.py`

y luego corregimos el método `get_queryset` de la siguiente manera: def get_queryset(self): """ Return the last five published questions (not including those set to be published in the future). """" return Question.objects.filter(pub_date__lte=timezone.now()).order_by('-pub_date')[:5]
`polls/views.py`

`Question.objects.filter(pub_date__lte=timezone.now)` devuelve un queryset que contiene las instancias de `Question` cuyo campo `pub_date` es menor o igual que - esto es, anterior o igual a - `timezone.now`.

Testeando nuestra nueva view

Ahora podemos verificar que se comporta como esperamos levantando el servidor de desarrollo, cargando el sitio en el browser, creando `Questions` 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 anterior con el `shell`.

Agregamos lo siguiente a `polls/tests.py`: from django.core.urlresolvers import reverse
`polls/tests.py`

vamos a crear un método de atajo para crear preguntas, como así también una nueva clase de test: def create_question(question_text, days): """ Creates a question with the given 'question_text' published the given number of 'days' offset to now (negative for questions published in the past, positive for questions that have yet to be published). """" time = timezone.now() + datetime.timedelta(days=days) return Question.objects.create(question_text=question_text, pub_date=time)

class QuestionViewTests(TestCase): def test_index_view_with_no_questions(self): """ If no questions 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_question_list’], [])

def test_index_view_with_a_past_question(self): “”” Questions with a pub_date in the past should be displayed on the index page. “”” create_question(question_text=”Past question.”, days=-30) response = self.client.get(reverse(‘polls:index’)) self.assertQuerysetEqual(response.context[’latest_question_list’], [’<Question: Past question.>’])

def test_index_view_with_a_future_question(self): “”” Questions with a pub_date in the future should not be displayed on the index page. “”” create_question(question_text=”Future question.”, days=30) response = self.client.get(reverse(‘polls:index’)) self.assertContains(response, “No polls are available.”, status_code=200) self.assertQuerysetEqual(response.context[’latest_question_list’], [])

def test_index_view_with_future_question_and_past_question(self): “”” Even if both past and future questions exist, only past questions should be displayed. “”” create_question(question_text=”Past question.”, days=-30) create_question(question_text=”Future question.”, days=30) response = self.client.get(reverse(‘polls:index’)) self.assertQuerysetEqual(response.context[’latest_question_list’], [’<Question: Past question.>’])

def test_index_view_with_two_past_questions(self): “”” The questions index page may display multiple questions. “”” create_question(question_text=”Past question 1.”, days=-30) create_question(question_text=”Past question 2.”, days=-5) response = self.client.get(reverse(‘polls:index’)) self.assertQuerysetEqual(response.context[’latest_question_list’], [’<Question: Past question 2.>’, ‘<Question: Past question 1.>’])
polls/tests.py

Vamos a mirarlo más detenidamente.

Primero tenemos una función, `create_question`, para evitar la repetición en el proceso de crear preguntas.

`test_index_view_with_no_questions` no crea preguntas, pero chequea el mensaje “No polls are available.” y verifica que `latest_question_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_question`, creamos una pregunta y verificamos que aparece en el listado.

En `test_index_view_with_a_future_question`, creamos una pregunta con `pub_date` en el futuro. La base de datos se resetea para cada método de test, entonces la primera pregunta 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 preguntas 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: class DetailView(generic.DetailView): ... def get_queryset(self): “”” Excludes any questions that aren’t published yet. “”” return Question.objects.filter(pub_date__lte=timezone.now())
polls/views.py

Y por supuesto, agregaremos más tests para chequear que una `Question` cuya `pub_date` es en el pasado se puede ver, mientras que una con `pub_date` en el futuro, no: class QuestionIndexDetailTests(TestCase): def test_detail_view_with_a_future_question(self): “”” The detail view of a question with a pub_date in the future should

return a 404 not found. “”” future_question = create_question(question_text=’Future question.’, days=5) response = self.client.get(reverse(‘polls:detail’, args=(future_question.id,))) self.assertEqual(response.status_code, 404)

def test_detail_view_with_a_past_question(self): “”” The detail view of a question with a pub_date in the past should display the question’s text. “”” past_question = create_question(question_text=’Past Question.’, days=-5) response = self.client.get(reverse(‘polls:detail’, args=(past_question.id,))) self.assertContains(response, past_question.question_text, status_code=200)

polls/tests.py

Ideas para otros tests

Deberíamos agregar un método `get_queryset` similar al de `ResultsView` y crear una nueva clase para los tests de esta view. Sería parecido a lo que hemos hecho ya; de hecho, habría bastante repetición.

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

Quizás usuarios administradores logueados deberían poder ver preguntas no publicadas, pero los demás usuarios no. 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:

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 `Questions` 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

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.

Testing en Django tiene información exhaustiva sobre testing.

Qué sigue?

Para un detalle completo sobre testing, ver Testing en Django.

Una vez que estés cómodo con el testing de views en Django, podés leer la *parte 6 de este tutorial* para aprender sobre el manejo de recursos estáticos.

1.1.8 Escribiendo tu primera Django app, parte 6

Este tutorial empieza donde dejó el *Tutorial 5*. Hemos construido una aplicación web de preguntas, y ahora vamos a agregar una hoja de estilos y una imagen.

Además del HTML generado por el servidor, las aplicaciones web en general necesitan servir archivos adicionales — como imágenes, JavaScript o CSS — necesarios para renderizar la página web completa. En Django nos referimos a estos archivos como "static files" (archivos estáticos).

Para proyectos pequeños esto no es un gran problema, porque uno mantiene los archivos estáticos en algún lugar que el servidor web los pueda encontrar. Sin embargo, en proyectos grandes – especialmente aquellos compuestos por múltiples apps – manejar múltiples conjuntos de archivos estáticos provistos por cada aplicación empieza a volverse complejo.

Para esto está `django.contrib.staticfiles`: reúne los archivos estáticos de cada una de nuestras aplicaciones (y cualquier otro lugar que uno especifique) en una única ubicación que pueda ser fácilmente servida en producción.

Personalizando el look and feel de nuestra *app*

Primero, creamos un directorio llamado `static` en nuestro directorio `polls`. Django va a buscar archivos estáticos allí, de forma similar a como Django busca por templates dentro de `polls/templates/`.

El setting de Django `STATICFILES_FINDERS` contiene una lista de los "finders" (buscadores) que saben cómo encontrar los archivos estáticos en distintas fuentes. Uno de los habilitados por defecto es `AppDirectoriesFinder` que busca un subdirectorio `static` en cada una de las `INSTALLED_APPS`, como la app `polls` que hemos creado. El sitio de admin usa la misma estructura de directorios para sus archivos estáticos.

Dentro del directorio `static` que hemos creado, creamos otro directorio llamado `polls` y dentro del mismo, un archivo llamado `style.css`. En otras palabras, nuestra hoja de estilos estaría en

`polls/static/polls/style.css`. Por como funciona `AppDirectoriesFinder`, nos podemos referir a este archivo estático en Django simplemente como `polls/style.css`, de forma similar a como referenciábamos templates.

Espacio de nombres de archivos estáticos

Como con los templates, *podríamos* poner todos nuestros archivos estáticos directamente en `polls/static` ``(en lugar de crear un subdirectorio ``polls`), pero esto no sería una buena idea. Django va a elegir el primer archivo estático que encuentre cuyo nombre coincida, y si tuviéramos un archivo del mismo nombre en otra aplicación *diferente*, Django no podría distinguir entre ellos. Necesitamos apuntar a Django al correcto, y la forma más fácil de asegurarse de esto es usando espacios de nombres. Esto es, poner los archivos estáticos en otro directorio que se llame como la aplicación a la que pertenecen.

Pongamos el siguiente código en la hoja de estilos (`polls/static/polls/style.css`): li a { color: green; }

`polls/static/polls/style.css`

Luego, agreguemos lo siguiente al comienzo de `polls/templates/polls/index.html`: {% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />

`polls/templates/polls/index.html`

{% load staticfiles %}‘ carga el template tag `{% static %}` de la librería `staticfiles`. El template tag `{% static %}` genera las URL absolutas del archivo estático.

Eso es todo lo que necesitamos para desarrollar. Recargamos `http://localhost:8000/polls/` y deberíamos ver que los links a las preguntas son verdes (Django syle!), lo que significa que la hoja de estilos se cargó correctamente.

Agregando una imagen de fondo

A continuación vamos a crear un subdirectorio para imágenes. Creamos el subdirectorio `images` en el directorio `polls/static/polls/`. Dentro de ese directorio ponemos una imagen llamada `background.gif`. En otras palabras, ponemos nuestra imagen en `polls/static/polls/images/background.gif`.

Luego agreguemos a nuestra hoja de estilos (`polls/static/polls/style.css`): body { background: white url("images/background.gif") no-repeat right bottom; }

`polls/static/polls/style.css`

Recargamos `http://localhost:8000/polls/` y deberíamos ver la imagen de fondo cargada en la esquina inferior derecha de la pantalla.

Advertencia: Por supuesto que el template tag `{% static %}` no está disponible para usar en los archivos estáticos ya que las hojas de estilo no son generadas por Django. Siempre deberías usar *paths relativos* para referenciar tus archivos estáticos entre sí, porque de esa forma uno puede cambiar el setting `STATIC_URL` (usado por el template tag `static` para generar las URLs), sin tener que modificar también una gran cantidad de paths en nuestros archivos estáticos.

Estos son los **conceptos básicos**. Para más detalles sobre los settings y otros bits incluidos en el framework, ver el howto de static files y la referencia de staticfiles. Deploying static files discute cómo usar archivos estáticos en un servidor real.

Qué sigue?

El tutorial para principiantes termina acá. 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*.

1.1.9 Tutorial avanzado: Cómo escribir apps reusables

Este tutorial avanzado comienza donde dejó el *Tutorial 6*. 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.

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.

Nuestro proyecto y nuestra app reusable

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

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

```
        urls.py
        wsgi.py
    polls/
        __init__.py
        admin.py
        migrations/
            __init__.py
            0001_initial.py
        models.py
        static/
            polls/
                images/
                    background.gif
                style.css
        templates/
            polls/
                detail.html
                index.html
                results.html
        tests.py
        urls.py
        views.py
    templates/
        admin/
            base_site.html
```

Creamos `mysite/templates` en el *Tutorial 2*, y `polls/templates` en el *Tutorial 3*. Ahora quizás es más claro por qué elegimos tener directorios separados de templates para el proyecto y la aplicación: todo lo que es parte de la app polls está en `polls`. Esto permite que la aplicación esté auto-contenido y sea más fácil reusarla en otro proyecto.

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

Instalando algunos prerrequisitos

El estado actual del empaquetado en Python está un poco confuso con varias herramientas. Para este tutorial vamos a usar setuptools para construir nuestro paquete. Es la herramienta recomendada (unida al fork de `distribute`). Vamos a usar también pip para instalarlo y desinstalarlo. Deberías instalar estos dos paquetes ahora. Si necesitaras ayuda, podés chequear cómo instalar Django con pip. De la misma manera se puede instalar `setuptools`.

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.

El nombre de una aplicación (es decir, la parte final del path de importación) *debe* ser único en `INSTALLED_APPS`. Evitá usar el mismo nombre que cualquiera de los paquetes contrib de Django, como `auth`, `admin`, o `messages`.

---

2. Movemos el directorio `polls` dentro de `django-polls`.

3. Creamos un archivo `django-polls/README.rst` con el siguiente contenido: ===== Polls =====

   Polls is a simple Django app to conduct Web-based polls. For each question, visitors can choose between a fixed number of answers.

   Detailed documentation is in the "docs" directory.

   Quick start ————

   1. Add "polls" to your INSTALLED_APPS setting like this::

   INSTALLED_APPS = (... 'polls',)

   2. Include the polls URLconf in your project urls.py like this::

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

   3. Run 'python manage.py migrate' to create the polls models.

   4. Start the development server and visit http://127.0.0.1:8000/admin/ to create a poll (you'll need the Admin app enabled).

   5. Visit http://127.0.0.1:8000/polls/ to participate in the poll. `django-polls/README.rst`

4. Creamos un archivo `django-polls/LICENSE`. Elegir una licencia está más allá del alcance de este tutorial, pero vale decir que código liberado públicamente sin una licencia es *inútil*. Django y muchas aplicaciones Django se distribuyen bajo la licencia BSD; sin embargo, uno puede elegir su propia licencia, teniendo en cuenta que ésta afecta quién puede utilizar 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 setuptools tiene una buena explicación. Creamos el archivo `django-polls/setup.py` con el siguiente contenido: import os from setuptools import setup

   with open(os.path.join(os.path.dirname(__file__), 'README.rst')) as readme: README = readme.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', # Replace these appropriately if you are stuck on Python 2. 'Programming Language :: Python :: 3', 'Programming Language :: Python :: 3.2', 'Programming Language :: Python :: 3.3', 'Topic :: Internet :: WWW/HTTP', 'Topic :: Internet :: WWW/HTTP :: Dynamic Content',],) `django-polls/setup.py`

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 include README.rst recursive-include polls/static * recursive-include polls/templates *

`django-polls/MANIFEST.in`

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 empaquetar, podés ver el Tutorial sobre empaquetar y distribuir proyectos.

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 una librería de usuario. Las instalaciones por usuario tienen muchas ventajas sobre las instalaciones a nivel sistema, como por ejemplo permitir usar un paquete en sistemas donde no se tiene acceso de administrador o también prevenir que un paquete afecte servicios de sistema y/o otros usuarios en la máquina.

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. Para instalar el paquete usamos pip (ya lo instalamos, no?):

   ```
   pip install --user django-polls/dist/django-polls-0.1.tar.gz
   ```

2. Con suerte, nuestro proyecto Django debería funcionar correctamente de nuevo. Levantamos el servidor de desarrollo para confirmarlo.

3. Para desinstalar el paquete, usamos pip:

   ```
   pip uninstall django-polls
   ```

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.
- Subir el paquete a un repositorio público, como ser Python Package Index (PyPI). packaging.python.org tiene un buen tutorial para esto.

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.

1.1.10 Qué leer a continuación

Bien, ya leíste el *material de introducción* y decidiste que querés seguir usando Django Esta introducción sólo cubre Django de manera superficial (de hecho, si leíste toda y cada una de las palabras de este material, es alrededor de un 5 % del total de la documentación).

Cómo seguir?

Bueno, siempre hemos sido grandes fans de aprender haciendo. En este punto deberías saber lo suficiente para empezar un proyecto por tu cuenta y empezar a probar. A medida que vayas necesitando trucos nuevos, podés volver a la documentación.

Hemos puesto un esfuerzo importante para hacer que la documentación de Django sea útil, fácil de leer y tan completa como sea posible. El resto de este documento explica un más sobre cómo funciona la documentación para que puedas aprovecharla al máximo.

(Sí, esta documentación es sobre la documentación. Podés quedarte tranquilo que no tenemos planes de escribir un documento sobre cómo leer el documento sobre la documentación).

Encontrando documentación

Django tiene *mucha* documentación – casi 450.000 palabras y contando –, entonces encontrar lo que uno necesita puede resultar complicado a veces. Un par de buenos lugares para empezar son la búsqueda y el índice general.

O navegarla!

Cómo se organiza la documentación

La documentación principal de Django se divide en “chunks”, diseñados para cubrir diferentes necesidades:

- El *material introductorio* está diseñado para la gente nueva en Django – o nueva en desarrollo web en general. No cubre nada en profundidad, pero en cambio da un vistazo general a cómo se siente desarrollar en Django.
- Las guías temáticas, por otro lado, dan un detalle más profundo de las partes individuales de Django. Hay guías completas sobre el sistema de modelos, el motor de templates, el framework de forms, y mucho más.

  Aquí es donde probablemente pases la mayor parte del tiempo; si le encontrás la vuelta a estas guías deberías salir sabiendo prácticamente todo lo que hay que saber de Django.
- El desarrollo web se extiende en general a lo ancho, no en profundo – los problemas se extienden a muchos dominios. Hemos escrito un conjunto de how-to guides que responden las preguntas comunes de la forma “How do I ...?”. Allí encontrarás información sobre generar PDFs con Django, escribir template tags propios, y más.

  Respuestas a preguntas realmente comunes se pueden encontrar en las FAQ.

- Estas guías y how-to's no cubren toda clase, función y método disponible en Django – eso sería demasiado cuando uno está intentando aprender. En vez de eso, los detalles de clases, funciones, métodos y módulos individuales se mantienen en la referencia. Allí es donde deberías dirigirte para encontrar los detalles particulares de una función o cualquier otra cosa que necesites.
- Si estás interesado en deployar un proyecto para su uso público, nuestra documentación tiene varias guías para distintas configuraciones, así como una checklist para algunos puntos a tener en cuenta.
- Finalmente, hay documentación "especializada" que no es relevante para la mayoría de los desarrolladores. Esto incluye release notes y internals documentation para aquellos que quieren contribuir código a Django mismo, y algunas otras cosas que no encajan en las demás secciones.

Cómo se actualiza la documentación

Así como el código base de Django se desarrolla y mejora de forma diaria, la documentación se mejora continuamente. Mejoramos la documentación por varias razones:

- Para hacer correcciones de contenido, tales como correcciones de tipeo o gramaticales.
- Para agregar información y/o ejemplos a secciones existentes que lo necesiten.
- Para documentar características de Django que todavía no lo estuvieran (esa lista de acorta, pero todavía hay cosas por documentar).
- Para agregar documentación de características nuevas a medida que éstas se agregan, o a medida que el comportamiento de las APIs de Django cambia.

La documentación de Django se mantiene en el mismo sistema de control de versiones que su código. Vive en el directorio docs de nuestro repositorio Git. Cada documento es un archivo de texto separado en el repositorio.

Dónde obtenerla

Podés leer la documentación de Django en varias formas. En orden de preferencia:

En la web.

La versión más reciente de Django vive en https://docs.djangoproject.com/en/dev/. Estas páginas HTML se generan automáticamente a partir de los archivos de texto en el repositorio. Esto significa que reflejan lo "último" en Django – incluyen las correcciones y adiciones más recientes, y discuten las más recientes características de Django, que puede que sólo estén disponibles a usuarios de la versión en desarrollo de Django (ver "Diferencias entre versiones" más abajo).

Te alentamos a ayudar mejorar la documentación, enviando cambios, correcciones y sugerencias en el sistema de tickets. Los desarrolladores de Django monitorean activamente el sistema de tickets y usan el feedback para mejorar la documentación para todos.

Notar sin embargo que los tickets deben ser relacionados explícitamente a la documentación, en lugar de hacer preguntas de soporte. Si necesitás ayuda con algo particular del uso de Django, probá en vez django-users mailing list o el canal de #django en IRC.

En texto plano

Para leer offline, o por conveniencia, se puede leer la documentación de Django en texto plano.

Si estás usando un release oficial de Django, notar que el paquete zip (tarball) del código incluye un directorio `docs/`, que contiene toda la documentación de ese release.

Si estás usando una versión en desarrollo de Django (aka "trunk"), notar que el directorio `docs/` contiene toda la documentación. Podés actualizar tu checkout para recibir los últimos cambios.

Una manera low-tech de aprovechar la documentación en texto es usando la utilidad `grep` de Unix para buscar una frase en toda la documentación. Por ejemplo, esto va a mostrar toda mención de la frase "max_length" en cualquier documento de Django:

```
$ grep -r max_length /path/to/django/docs/
```

Como HTML, localmente

Se puede obtener una copia del HTML de la documentación siguiendo unos simples pasos:

- La documentación de Django usa un sistema llamado Sphinx para converitr de texto plano a HTML. Vas a necesitar instalar Sphinx ya sea bajando e instalando el paquete del sitio web de Sphinx, o con `pip`:

  ```
  $ pip install Sphinx
  ```

- Luego, basta usar el archivo `Makefile` incluido para generar la documentación en HTML:

  ```
  $ cd path/to/django/docs
  $ make html
  ```

  Vas a necesitar GNU Make instalado para esto.

  Si estás en Windows, podés alternativamente usar el archivo batch incluido:

  ```
  cd path\to\django\docs
  make.bat html
  ```

- La documentación HTML se ubicará en `docs/_build/html`.

Nota: La generación de la documentación de Django funciona con Sphinx 0.6 o mayor, pero recomendamos usar 1.0.2 o más nueva.

Diferencias entre versiones

Como se mencionó antes, la documentación en texto en el repositorio Git contiene "lo último" en cambios y adiciones. Estos cambios a veces incluyen documentación de nuevas características que se agregan en la version en desarrollo de Django – la versión Git ("trunk") de Django. Por esta razón vale la pena mencionar nuestra política de mantener la documentación al día para las distintas versiones del framework.

Seguimos esta política:

- La documentación primaria en djangoproject.com es la version HTML de lo último en Git. Esta documentación se corresponde con la última release oficial de Django, además de las características que se agregaron/cambiaron *desde* el último release.
- A medida que se agregan características a la versión en desarrollo de Django, tratamos de mantener la documentación actualizada en el mismo commit de Git.
- Para distinguir cambios/adiciones en la documentación, se usa la frase "New in version X.Y", donde X.Y es la próxima versión a ser liberada (es decir, la versión en desarrollo).
- Correcciones y mejoras a la documentación se portan a las versiones anteriores a discreción del committer; sin embargo, una vez que la versión de Django no se soporta más, la documentación de esa versión no recibe más actualizaciones.

- La pagína principal de la documentación en la web incluye links a la documentación de todas las versiones anteriores. Fijate de chequear que la versión de la documentación sea la que corresponde a la versión de Django que estás usando!

1.1.11 Escribiendo tu primer parche para Django

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.

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.

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.

Volviendo a una revisión anterior de Django

Para este tutorial vamos a usar el #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
```

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.

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.

Escribir el código para el ticket

A continuación vamos a agregar a Django la funcionalidad descripta en el #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">{} <a {}>{}</a><br />{} {}</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 3 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.

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.

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 la sección correspondiente. 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.

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">{} <a {}>{}</a><br />{} {}</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 />Chang
+        )
+
```

```
+    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.c
+        )
+
+    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/someta
+        )
+        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%20t
+        )
```

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

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.

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. Si este libro no sigue tu estilo, hay algunos otros libros sobre Python.

Documentación completa (en inglés): https://www.docs.djangoproject.com/

D

V