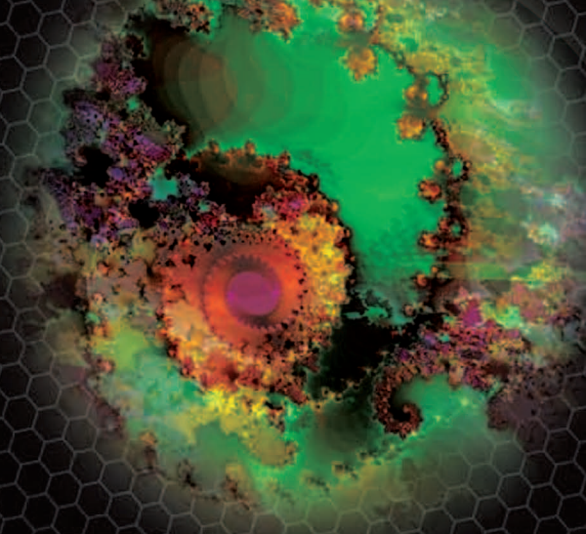


Introducción a

django

Escribe tu primera aplicación Web usando Django

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

Saúl García M.

Django Software Corporation

Tutorial de introducción a Django: Escribe tu primera aplicación Web usando Django 1.8

El código fuente en formato RST para este libro y más información sobre este proyecto se encuentra en el repositorio:

http://github.com/saulgm/djangobook.com

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

Todos estos son programas de código abierto y gratuitos.

Ideas, sugerencias; quejas: saulgarciamonroy@gmail.com
Publicado, editado y compilado: en algún lugar de Celayita México.

Introducción a Django

Escribe tu primera aplicación Web usando Django 1.8

Saul Garcia M.

Revisión 1.8

Django Software Corporation

Capítulos

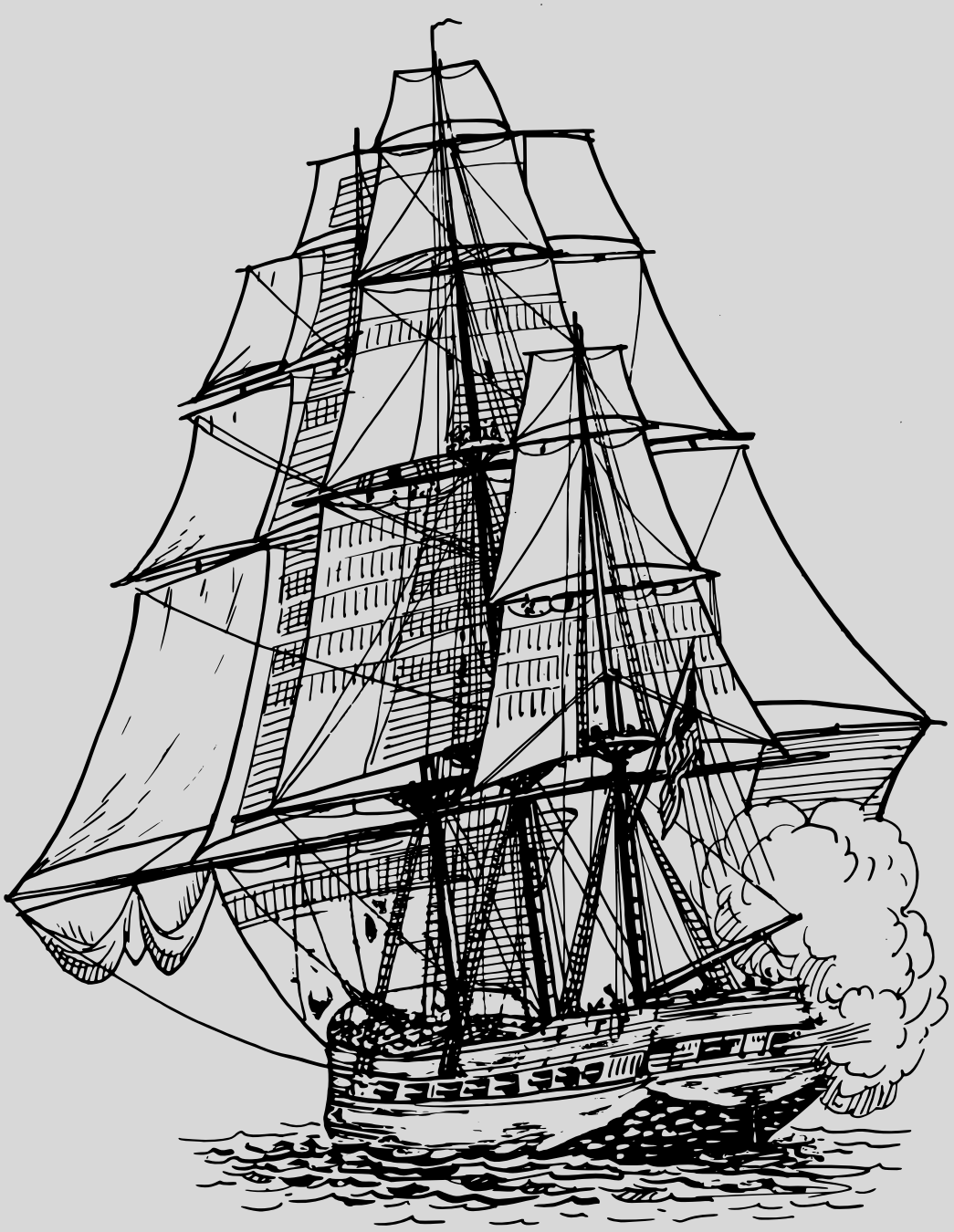
PARTE 1 Tutorial de introducción a Django

Contenido

PARTE 1 Tutorial de introducción a Django

PRIMERA PARTE

Iniciando con Django



CAPÍTULO 01

Introducción a Django

En sus inicios, Django fue desarrollado como una solución práctica para un entorno de noticias Web, fue diseñado para hacer las tareas comunes y repetitivas del desarrollo Web de forma rápida y sencilla.

Debido a que Django fue extraído de código de la vida real, en lugar de ser un ejercicio académico o un producto comercial, está especialmente enfocado en resolver problemas de desarrollo Web con los que los desarrolladores de Django se han encontrado – y con los que continúan encontrándose. Como resultado de esto, Django es continuamente mejorado.

Esta es una pequeña descripción informal sobre cómo escribir aplicaciones Web, con una base de datos manejada por Django.

La meta de esta tutorial es proporcionarte información no tan técnica, para poder entender cómo funciona Django, solamente eso no pretende ser una guía de referencia; ¡pero tenemos ambas cosas! Cuando estés listo para comenzar un proyecto Web con Django, puedes leer el tutorial de introducción a Django para aprender a usar Django o puedes leer el libro de Django, para conocer cuestiones más técnicas y completas.

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

Si no tienes experiencia programando en Python, te espera una grata sorpresa. Python es fácil de aprender y muy divertido de usar, es por ello que incluimos un *tutorial completo* en el que se hace hincapié en las características y funcionalidades de Django.

Si tienes experiencia programando en Python, no deberías tener problemas en entender cómo funciona Django, ya que el código no produce "magia negra", ni trucos de programación cuya implementación sea difícil de explicar o entender.

Por lo que aprender a utilizar Django, es sólo cuestión de aprender las convenciones y APIs que implementa en Python.

Diseña tu modelo

Aunque es posible usar Django sin una base de datos, este incluye por defecto un conveniente mapeador relacional (http://en.wikipedia.org/wiki/Object-relational_mapping) en el que es posible describir la estructura de la base de datos usando solamente código Python.

La sintaxis, para describir el modelo de datos ofrece muchas formas de representar los modelos, en el que se han resuelto diversos problemas con los esquemas de las bases de datos por varios años, lo que ha redundado en una API de modelos estable y sencilla de usar.

El siguiente es un ejemplo rápido:

misitio/noticias/models.py

```
from django.db import models

class Reportero(models.Model):
    nombre_reportero = models.CharField(max_length=70)

    def __str__(self):              # __unicode__ en Python 2
        return self. nombre_reportero

class Articulo(models.Model):
    fecha = models.DateField()
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    reportero = models.ForeignKey(Reportero)

    def __str__(self):              # __unicode__ en Python 2
        return self.titulo
```

Instálalo

Con el modelo listo, ejecuta la utilidad de línea de comandos `manage.py` para crear la base de datos automáticamente:

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

El comando migrate busca todos los modelos disponibles de cada una de las aplicaciones registradas y crea las tablas correspondientes en la base de datos si es que estas no existían previamente, con lo que opcionalmente se ofrecen un mayor control sobre el esquema de la base de datos.

Juega con la API

Con esto hemos conseguido, crear una base de datos, sin mayor esfuerzo, ahora podemos acceder a los datos gracias a una expresiva y potente API creada en Python. La API se crea al vuelo, sin necesidad de generar código, ahora podemos acceder a ella mediante una terminal:

```
# Importamos  los modelos de nuestra aplicación
>>> from noticias.models import Reportero, Articulo
# Aun no  tenemos reporteros en el sistema.
>>> Reportero.objects.all()
[]
# Creamos un nuevo Reportero.
>>> r = Reportero(nombre_reportero='John Smith')
#  Guardamos  el  objeto  en  la  base  de  datos.  Llamando
explícitamente a  save()
>>> r.save()
# Accedemos al identificador ID del objeto guardado.

>>> r.id
```

```
1
# Ahora tenemos un reportero en la base de datos.
>>> Reportero.objects.all()
[<Reporter: John Smith>]
# Los campos son representados como atributos de objetos Python.
>>> r.nombre_completo
'John Smith'
# Django provee una rica API, para realizar consultas.
>>> Reportero.objects.get(id=1)
<Reporter: John Smith>
>>> Reportero.objects.get(nombre_reportero__startswith='John')
<Reporter: John Smith>
>>> Reportero.objects.get(nombre_reportero__contains='mith')
<Reporter: John Smith>
>>> Reportero.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist.

# Creamos un articulo
>>> from datetime import datetime
>>> a =Articulo(fecha=datetime.now(), titulo='Django es genial',
...     contenido='Yeah.', reportero=r)
>>> a.save()
# Ahora el articulo está en la base de datos.
>>> Articulo.objects.all()
[<Articulo: Django es genial>]
# Podemos acceder a los objetos relacionados a través de la API.
>>> r = a.reportero
>>> r.nombre_reportero
'John Smith'
# Y viceversa: Los Reporteros pueden acceder mediante la  API
# a los Artículos.
>>> r.articulo_set.all()
[<Articulo: Django es genial>]
# La API crea relaciones tan lejos como la necesites.
# Trabaja detrás de escena.
# Encuentra todos los artículos de un reportero cuyo
# nombre empieza con "John".
>>>
Articulo.objects.filter(reportero__nombre_reportero__startswith=
"John")
[<Articulo: Django es genial>]
# Cambiamos un objeto alterando sus atributos y llamando
# al método save().
>>> r.nombre_reportero = 'Billy Goat'
>>> r.save()
# Borramos un  objeto con el método  delete().
>>> r.delete()
```

La interfaz de administración

Una interfaz de administración dinámica, no es sólo la estructura de la casa; es la casa completa. Una vez que los modelos están definidos, Django crea automáticamente una interfaz de administración profesional y lista para producción, un sitio Web que permite a los usuarios autentificados agregar, cambiar y eliminar contenidos. Es tan fácil como registrar el modelo en el sitio de administración así:

misitio/noticias/admin.py

```
from django.contrib import admin

from . import models

admin.site.register(models.Articulo)
admin.site.register(models.Reportero)
```

La filosofía aquí es que el sitio pueda ser modificado por personal administrativo, por un cliente o tal vez por el mismo desarrollador, por lo que no hay necesidad de preocuparse por crear interfaces de administración sólo para gestionar contenidos.

Un típico flujo de trabajo en la creación de aplicaciones Django, consiste en crear modelos y habilitar el sitio de administración tan rápido como sea posible, de forma que el personal (o los clientes) puedan comenzar a introducir datos. Luego, desarrollar la forma en que los datos son presentados al público.

Diseña tus URL

Un esquema de URLs limpio y elegante es un detalle importante en una aplicación web de alta calidad. Django incentiva el diseño de URLs elegantes y no agrega ningún lastre a las URLs, como .php o .asp.

Para diseñar las URLs de tu aplicación, se crea un módulo Python llamado URLconf, que es como una tabla de contenidos para las aplicaciones que contiene, un mapeo simple entre patrones de URLs y funciones Python. Las URLconfs también sirven para desacoplar las URLs del código Python.

Así es como quedaría la URLconf para el ejemplo presentado anteriormente:

misitio/noticias/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articulos/([0-9]{4})/$', views.archivos_anuales),
    url(r'^articulos/([0-9]{4})/([0-9]{2})/$',
        views.archivos_mensuales),
    url(r'^articulos/([0-9]{4})/([0-9]{2})/([0-9]+)/$',
        views.detalle_articulo),
]
```

Este código asocia las URLs, como simples expresiones regulares o regex (http://docs.python.org/2/howto/regex.html), con funciones Python (“vistas”). Las expresiones regulares usan paréntesis para “capturar” valores de las URLs.

Cuando un usuario solicita una página, Django verifica cada patrón, en orden y se detiene en el primero que coincida con la URL solicitada. Si ninguno coincide, Django llama a una vista especial llamada 404, que no es otra cosa que una "*pagina no encontrada*". Esto es increíblemente rápido, porque las expresiones regulares son compiladas cuando se carga el código.

Una vez que una de las expresiones coincide, Django importa y llama la vista correspondiente, la cual es una simple función Python. Cada vista recibe un objeto `request` que contiene los metadatos de la petición y los valores capturados en la expresión regular.

Por ejemplo, si un usuario solicita la URL "/articulos/2015/01/39323/", Django llamaría a la función noticias. detalle_articulo (request, "2015", "01", "39323").

Escribe tus vistas

Cada vista es responsable de hacer una de dos cosas: Devolver un objeto HttpResponse con el contenido de la página solicitada o lanzar una excepción como Http404. Lo demás es responsabilidad del desarrollador.

Generalmente, una vista recupera datos de acuerdo a los parámetros que le hemos pasado, primero carga la plantilla y la rellena con los datos recuperados. Aquí hay una vista para el ejemplo anterior: archivos_anuales:

misitio/noticias/views.py

```
from django.shortcuts import render

from .models import Articulo

def archivos_anuales(request, year):
    lista_articulos = Articulo.objects.filter(fecha__year=year)
    contexto = {'year': year,  'lista_articulos':
        lista_articulos}

    return render(request,  'noticias/archivos_anuales',
        contexto)
```

Este ejemplo usa el sistema de plantillas de Django, este sistema posee varias características poderosas y al mismo tiempo es lo suficientemente simple para poder ser usado por no programadores.

Diseña tus plantillas

El código anterior carga la plantilla *noticias/archivos_anuales.html*.

Django posee de forma predeterminada una ruta de búsqueda de plantillas, lo que permite minimizar la redundancia entre ellas, esta ruta se encuentra definida en el archivo de configuración del proyecto, donde es posible especificar una lista de directorios donde se buscarán las plantillas. Si una plantilla no existe en el primer directorio, se busca en el siguiente y en el siguiente y así sucesivamente.

Supongamos que se encontró la plantilla noticias/archivos_anuales. Aquí hay un ejemplo de cómo podría lucir esa plantilla:

misitio/noticias/templates/noticias/archivos_anuales.html

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

{% block title %}Articulos del {{ year }}{% endblock %}

{% block content %}
<h1> Articulos  del {{ year }}</h1>

{% for articulo  in lista_articulos %}
    <p>{{ articulo.titulo }}</p>
    <p>Por  {{ articulo.nombre_reportero }}</p>
    <p>Publicado el {{ articulo.fecha|date:"F j, Y" }}</p>
{% endfor %}

{% endblock %}
```

Las variables están encerradas por llaves dobles. {{ articulo.titulo}} significa "Mostrar el valor del atributo titulo de articulo". Pero los puntos no sólo son usados para buscar atributos, también sirven para buscar claves en diccionarios, para buscar índices y para hacer llamadas a funciones.

Observa que {{ articulo.fecha|date:"F j, Y" }} usa un "pipe" Unix (el carácter "|", también llamado tubería), que no es más que un simple filtro de plantilla y es una forma de filtrar el valor de una variable. En este caso, el filtro `date` da formato a un objeto datetime de Python (tal como ocurre en la función date de PHP)

Es posible encadenar tantos filtros como se deseen. Además se pueden codificar filtros a la medida. De la misma forma, es posible codificar etiquetas de plantillas a la medida, los cuales pueden ejecutar código Python en segundo plano.

Finalmente, Django usa el concepto de "herencia de plantillas": Esto es lo que hace {% extends "base.html" %}. "Primero carga la plantilla llamada 'base', la cual contiene algunos bloques previamente definidos, que serán rellenados con bloques de código resultantes de alguna consulta o alguna función Python. En otras palabras, permite disminuir dramáticamente la redundancia en las plantillas: Cada plantilla tiene que definir sólo lo que le es propio.

Así es como quedaría "base.html", incluyendo el uso de archivos estáticos (como el logotipo del sitio Web):

misitio/templates/base.html

```
{% load staticfiles %}

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

En pocas palabras, define el look-and-feel del sitio (como el logotipo) y provee espacios para ser rellenados por las plantillas hijas. Esto permite que el rediseño de un sitio sea tan fácil como cambiar un único archivo -- la plantilla base.

Además el sistema de plantillas, permite crear múltiples versiones de un sitio, con diferentes plantillas base, rehusando las plantillas hijas. Los creadores de Django han

usado esta técnica para crear ediciones para dispositivos móviles, notablemente distintos de algunos sitios – simplemente creando una nueva plantilla base.

Ten en cuenta que no es necesario usar el sistema de plantillas de Django, si es que prefieres usar otro sistema. Aunque el sistema de plantillas de Django está particularmente bien integrado con la capa de modelo de Django, no es obligatorio usarlo. Por la misma razón, tampoco es necesario usar la API de base de datos de Django. Es posible utilizar otra capa de abstracción de datos, también es posible leer archivos XML, leer archivos desde el disco o lo que se quiera. Cada pieza de Django – modelos, vistas, plantillas – está desacoplada del resto.

Esto es solo la superficie

Esta ha sido sólo una vista rápida a las funcionalidades de Django. Algunas otras características útiles que vale la pena mencionar son:

- Un framework de caché que se integra con memcached y otros sistemas similares.
- Un framework de sindicación que permite crear feeds RSS y Atom de manera tan fácil como crear pequeñas clases Python.
- GeoDjango Un framework para crear aplicaciones Web geográficas, tales como servicios de localización.
- Interfaces de administración generadas automáticamente más elegantes – esta introducción sólo toca la superficie del tema.

Los siguientes pasos son obvios: descarga Django, lee el tutorial y únete a la comunidad.

¡Gracias por tu interés!

CAPÍTULO 02

Guía de instalación rápida

Antes de que puedas utilizar Django, necesitas primero instalarlo. Esta es una pequeña guía de inicio rápido, no pretende ser una exhaustiva guía de instalación, que cubra todas las posibles instalaciones de Python y Django, si no simplemente una sencilla y rápida introducción que te permita instalar e iniciar Django.

Instala Python

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

El núcleo del framework trabaja bien con cualquier versión de Python superior a la 2.76. A partir de este lanzamiento Python 3 es oficialmente soportado. Es recomendable utilizar ampliamente las últimas y menores versiónes de cada lanzamiento de Python, por ejemplo: 2.7.X, 3.2.X, 3.3.X y 3.4.X.

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

Si decides utilizar la serie 3 de Python, puedes usar cualquiera de las versiones 3.2, 3.3 y 3.4. Estas versiones de Python incluyen una base de datos ligera llamada Sqlite, por lo que no será necesario configurar e instalar una base de datos para usar Django por el momento.

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

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

Instala una base de datos

Si instalaste una versión superior a Python 2.76, puedes saltarte este paso por ahora, sin embargo si quieres trabajar con un motor de base de datos “grande” como PostgreSQL, MySQL, u Oracle, puedes consultar la referencia rápida sobre instalación de bases de datos para obtener información más detallada sobre el proceso de instalación.

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

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

Instala Django

En esta sección se explican algunas opciones de instalación. En cualquier momento, puedes disponer de dos versiones distintas de Django para utilizar en tus proyectos:

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

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

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

Quita cualquier versión antigua de Django: Si estas actualizando tu vieja instalación de Django, necesitaras desinstalar la versión de Django anterior, antes de instalarte la nueva versión.

Si previamente instalaste Django usando python setup.py install, para desinstalarlo, simplemente borra el directorio de django situado en site-packages.

Instala un lanzamiento oficial

La mayoría de los usuarios querrán instalar el lanzamiento oficial más reciente desde la página Web del proyecto en: http://www.djangoproject.com/download/. Django usa el método distutils estándar de instalación de Python, que en el mundo de Linux es así:

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

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

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

Instalar un lanzamiento oficial usando pip

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

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

- Abre una terminal de comandos y ejecuta el comando easy_install pip. Este comando instalara pip en tu sistema (La versión 3.4 de Python, lo incluye como el instalador por defecto, asi que puedes saltarte estos pasos.).

Si estas usando Linux, Mac OS X o algún otro sabor de Unix, usa el siguiente comando en una terminal para instalar Django:

```
sudo pip install django
```

Si estas usando Windows, inicia el shell de comandos con privilegios de administrador y ejecuta el comando: pip install django.

Instalar la “Versión de Desarrollo”

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

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

Clona el repositorio usando el comando:

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

Verifica tu instalación

Para verificar que Django está instalado correctamente, cámbiate a algún otro directorio e inicia python en una terminal. Si todo está funcionando bien, deberías poder importar el módulo django así:

```
>>> import django
>>> print(django.get_version())
1.8
```

¡Eso es todo!

Eso es todo lo que necesitas para empezar - Ahora puedes iniciar el *tutorial de introducción a Django,* para iniciar un proyecto.

CAPÍTULO 03

Tu primera aplicación Django, parte 01

A prendamos mediante un ejemplo.

A lo largo del siguiente tutorial, analizaremos paso a paso, la creación de una aplicación básica de encuestas Web, usando Django.

Esta aplicación constará de dos partes:

- Un **sitio público** que permitirá a las personas ver las encuestas y votar en ellas.
- Un **sitio administrativo** que permitirá añadir, cambiar y suprimir las encuestas.

Asumiremos que ya has instalado Django, esto se puede verificar rápidamente ejecutando en una terminal el comando python y tecleando *import django.* Si el comando se ejecuta exitosamente sin errores, entonces Django está instalado:

```
>>> import django
>>> django.VERSION
1.8
>>>
```

O puedes usar directamente el siguiente comando en una terminal:

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

Este tutorial está escrito para Django 1.8 y python 3.2 o superior, si la versión de Django que utilizas no corresponde con la de este tutorial, puedes buscar una versión apropiada o puedes actualizarte a esta versión si e tu caso. Si estas usando Python 2.7, debes ser consciente de que el código puede diferir del de este tutorial, pero puedes continuar usándolo, ajustando el código tal como se describe en los comentarios.

Puedes consultar "Como instalar Django" para buscar consejos acerca de como remover versiónes anteriores de Django, antes de instalar una nueva versión.

Donde obtener ayuda: Si te surgen problemas al seguir este tutorial, puedes enviar un mensaje a django-users o entra al canal de chat #django en irc.freenode.net donde otros usuarios de django te pueden ayudar.

Crea un proyecto

Si ésta es la primera vez que utilizas Django, tendrás que crear una configuración inicial. Es decir; necesitarás auto-generar código que defina un proyecto Django – que es un conjunto de configuraciones para una instancia de Django, incluyendo la configuración de una base de datos y algunas opciones específicas de Django y detalles propios de una aplicación.

Desde la línea de comandos, utiliza el comando cd para cambiarte de directorio a aquel donde quieras almacenar el código y teclea en una terminal:

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

Donde misitio es el nombre de tu proyecto. Esto creará un directorio misitio en el directorio actual, que contiene la estructura de directorios y archivos que conforman tu proyecto Django.

Nota: Evita que tu proyecto tenga un nombre igual al de algún componente interno de Django o de Python. En particular, debes evitar usar nombres como django (que entrará en conflicto con Django mismo) o test (que causa conflictos con un paquete interno de Python).

El nombre del script puede ser diferente en distintas distribuciones. Si usas Linux e instalaste Django usando un instalador de paquetes (e.g. yum o apt-get) django-admin.py este ha sido renombrado a django-admin. Si usas Windows la instalación creara un ejecutable llamado django-admin. Puedes continuar leyendo esta documentación omitiendo el .py de cada comando

---

django-admin.py debería estar en la ruta de búsqueda del sistema (system path) si instalaste Django usando python setup.py. Si no está en la ruta, lo puedes encontrar en site-packages/django/bin, donde esta site-packages, que es un directorio dentro de la instalación de Python. De otra forma considera crear un enlace simbólico hacia django-admin.py desde algún lugar en la ruta de búsqueda, por ejemplo /usr/local/bin.

¿Dónde deberías poner tu código? Si tienes experiencia usando otros lenguajes como PHP, probablemente estás acostumbrado a ubicar el código en la raíz de documentos del servidor web (algo como /var/www). Con Django no es así. No es una buena idea poner nada de este código en la ruta raíz del servidor web, porque se corre el riesgo de que alguien pueda ser capaz de ver el código a través de la Web. Eso no es bueno en términos de seguridad.

Ubica el código en algún directorio **fuera** de la raíz del servidor, por ejemplo en /home/proyecto.

Veamos lo que el comando startproject creo:

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

Estos archivos son:

- **misitio**/: Un directorio externo, que es justamente un contenedor para tu proyecto. Su nombre no le importa a Django; puedes renombrarlo con cualquier otro nombre.

- **manage.py**: La línea de comandos que te permite interactuar con un proyecto Django de muchas formas.

- **misitio/misitio/:** El directorio interno que contiene el paquete para tu proyecto Django. El nombre del directorio, es el nombre del paquete que python usa para importar cualquier cosa dentro de él (por ejemplo import misitio.settings).

- **misitio/__init__.py:** Un fichero vacío que le dice a python que este directorio se debe considerar como un paquete python. Puedes leer más sobre paquetes en (http://docs.python.org/tutorial/modules.html#packages) en la guía oficial de python si eres principiante.)

- **misitio/settings.py**: El archivo de configuraciones para tu proyecto Django, contiene todas las configuraciones necesarias para un proyecto Django.

- **misitio/urls.py**: La URL para este proyecto de Django; una tabla de contenidos para tu sitio Django.

- **misitio/wsgi.py**: El punto de entrada para el servidor Web. Un archivo compatible con el estándar WSGI para configurar el servidor Web, usando Python.

Configuración de la base de datos

Ahora, edita el archivo misitio/settings.py. Este archivo es un módulo normal de Python con variables a nivel de módulo que representan las configuraciones de Django.

Está configurado por defecto, para usar SQLite. Si eres nuevo utilizando bases de datos y estas solo interesado en probar Django, esta es la opción más sencilla de utilizar. SQLite se incluye en Python, así que no es necesario instalar nada, para dar soporte a una base de datos.

Si deseas utilizar otra base de datos, necesaritas primero instalar y configurar la base de datos de tu preferencia, instalando los conectores apropiados, puedes consultar “configurar una base de datos”, para obtener mayor información y cambia una de las siguientes claves de configuración DATABASES en la variable 'default':

- **ENGINE** – Usa una de las siguientes configuraciones: 'django.db.backends.sqlite3', 'django.db.backends.postgresql_psycopg2', 'django.db.backends.mysql', o 'django.db.backends.oracle'.

- **NAME** – El nombre de la base de datos. Si estas usando SQLite como base de datos, este será un archivo en tu ordenador; en tal caso, NAME deberá ser la ruta absoluta, incluyendo el nombre de el archivo. Si el archivo no existe se creará automáticamente cuando sincronices la base de datos por primera vez. El valor por defecto es os.path.join(BASE_DIR, 'db.sqlite3'), el cual almacena el archivo de base de datos en el directorio actual de tu proyecto.

Si no estás usando SQLite como base de datos, necesitas agregar datos adicionales tal como USER, PASSWORD, HOST, para más detalles consulta la documentación de referencia para encontrar una guía completa.

Nota: Si estás usando PostgreSQL o MySQL, debes asegurarte de crear una base de datos en este punto. Lo puedes hacer con el comando **"`CREATE DATABASE nombre_base_de_datos;`"** mediante el intérprete interactivo de la propia base de datos que vayas a utilizar.

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

En el mismo archivo *settings.py*, editamos la variable *TIME_ZONE,* las zonas horarias (http://en.wikipedia.org/wiki/List_of_tz_zones_by_name), el valor por defecto es la zona horaria central para los E.E.U.U. (Chicago), puedes cambiarlo por el de tu preferencia.

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

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

- django.contrib.admin – La interfaz administrativa. La cual usaremos en la siguiente parte de este *tutorial*

- django.contrib.auth – El sistema de autentificación.

- django.contrib.contenttypes – Un framework para tipos de contenidos.

- django.contrib.sessions – Un framework. para manejar sesiones

- django.contrib.messages – Un framework para manejar mensajes

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

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

Cada uno de estas aplicaciones hace uso, de por lo menos una tabla de la base de datos, por lo que necesitas crear las tablas las tablas antes de que puedas utilizarlas, para hacerlo ejecuta el comando siguiente:

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

El comando migrate busca la variable *INSTALLED_APPS* y crea las tablas necesarias de cada una de las aplicaciones registradas, de acuerdo a la configuración de la base de datos registrada en el archivo misitio/settings.py, que contiene todas las aplicaciones, la cual veremos más adelante. Veras un mensaje por cada migración aplicada. Si estas interesado, ejecuta en la línea de comandos el cliente de tu base de datos o usa directamente el comando python manage.py dbshell para iniciarlo, y usa \dt (PostgreSQL), SHOW TABLES; (MySQL), o .schema (SQLite) para mostrar las tablas que Django ha creado en la base de datos.

Para los minimalistas: Como mencionamos anteriormente, las aplicaciones por defecto son incluidas para los casos más comunes, pero no todas son necesarias. Si no quieres instalar alguna, simplemente comenta o elimina las líneas correspondientes en INSTALLED_APPS, solo ten en cuenta que algunas aplicaciones dependen de otras, antes de ejecutar el comando migrate. El comando migrate sólo migrara las tablas para las aplicaciones listadas en el archivo INSTALLED_APPS.

El servidor de desarrollo

Para verificar que Django trabaja, cámbiate al directorio del proyecto misitio, si es que no estabas allí y ejecuta el comando:

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

Veras una salida como la siguiente en tu terminal:

```
Performing system checks...

0 errors found
January 06, 2014 - 15:50:53
Django version 1.8, using settings 'misitio.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Acabas de iniciar el servidor de desarrollo de Django, un servidor web liviano escrito completamente en Python, que vine incluido en Django para poder desarrollar de manera rápida tus proyectos, sin tener que perder el tiempo con la configuración de un servidor de producción – como Apache hasta que el proyecto esté listo para producción.

Ahora que el servidor está funcionando dirígete a la página http://127.0.0.1:8000/ usando un navegador web. Verás una página "Welcome to Django", en un agradable tono azul claro. ¡Funcionó!

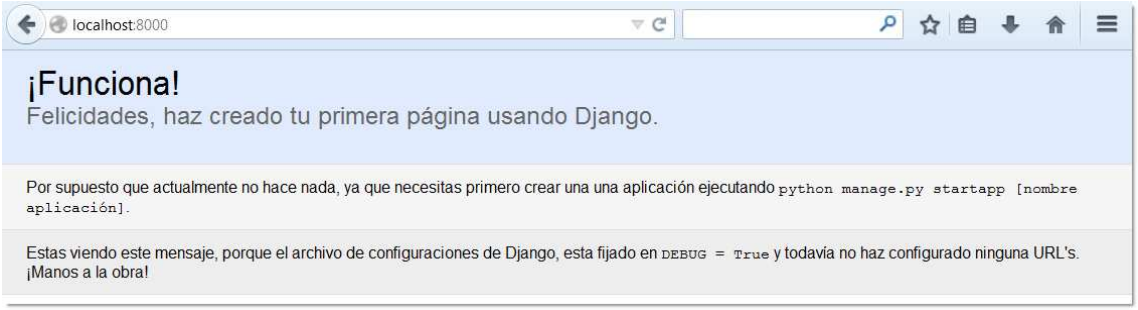


Imagen 01: *Pagina de bienvenida de Django*

Este es un buen momento para que recordarte que: **NO** debes usar este servidor Web, en nada que se parezca a un entorno de producción. Su objetivo es ser usado sólo para desarrollo de forma local.

Cambia el puerto

Por defecto el comando *runserver* inicia el servidor de desarrollo en la IP interna local en el puerto 8000. Si quieres cambiar el acceso al servidor, solo debes pasarle él puerto como argumento en la línea de comandos a *runserver*. Por ejemplo, este comando inicia el servidor en el puerto 8080:

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

Si quiere lanzar el servidor para trabajar con otras computadoras en una IP pública puedes usar el siguiente comando:

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

Puedes consultar la documentación completa, para obtener más información sobre el servidor de desarrollo.

Recarga automática de runserver

El servidor de desarrollo, automáticamente recarga el código Python para cada petición, según se necesite. No necesitas recargar el servidor para que los cambios al código tomen efecto, sin embargo, para algunas acciones como agregar o borrar archivos, es necesario volver a arrancar el servidor para que cargue el código en estos casos.

Creando los modelos

Ahora que hemos configurado las variables de entorno para nuestro "proyecto", estamos listos para comenzar a trabajar.

Cada aplicación que escribes en Django consiste de un paquete Python, en algún lugar de la ruta de búsqueda de Python (Python path,) que sigue una cierta convención. Django viene con una utilidad que genera automáticamente la estructura de directorios básica de una aplicación, de forma que nos podamos enfocar en escribir código, en vez de estar creando directorios.

¿Cuál es la diferencia entre un proyecto y una aplicación?

Una **aplicación** es una aplicación web que hace algo – por ejemplo, una bitácora, un registro de datos públicos o un sistema de encuestas. Un **proyecto** es un conjunto de aplicaciones configuradas para un sitio web particular. Un proyecto puede contener múltiples aplicaciones. Una aplicación puede pertenecer a múltiples proyectos.

Nuestras aplicaciones pueden estar ubicadas en cualquier lugar, siempre que estén en la ruta de búsqueda de python o en el Python path, En este tutorial crearemos una aplicación llamada ***encuestas***, en el mismo directorio donde se encuentra el comando manage.py, así el archivo pueda ser importado como un módulo python en un nivel superior o como un sub-módulo de misitio.

Para crear nuestra aplicación, debemos asegurarnos de estar en el mismo directorio donde está el archivo manage.py y con el comando siguiente, crearemos una aplicación llamada ***encuestas***:

```
$ python manage.py startapp encuestas
```

Esto creará un directorio encuestas, con una estructura de archivos como este:

```
encuestas/
    __init__.py
    admin.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

Esta estructura de directorios contiene la aplicación encuestas. El primer paso para escribir una base de datos en Django es definir sus modelos – esencialmente, un modelo es el diseño de la base de datos, con los meta datos adicionales.

Filosofía

Un modelo es la fuente única y definitiva de información de los datos. Contiene los campos y comportamiento esenciales de los datos que se están almacenando. Django

sigue el *Principio DRY* (Don't Repeat Yourself – No te repitas). El objetivo es definir el modelo de datos en un solo lugar y derivar cosas automáticamente a partir de él.

Esto incluye las migraciones - tal como en Ruby On Rails por ejemplo, las migraciones se derivan enteramente de los archivos de los modelos y son esencialmente el historial de cambios de los modelos, de esta forma Django pueda actualizar el esquema de la base de datos para que coincida con los modelos actuales.

En nuestro sistema de encuestas, crearemos dos modelos: ***Pregunta*** *y* ***Opción***. Pregunta contiene el texto de una pregunta y una fecha de publicación. Opcion contiene dos campos: el texto de la opción y un contador de votos. Cada opción está asociada a una pregunta.

Estos conceptos son representados por simples clases python. Editamos el fichero encuestas/models.py para que se parezca a este:

encuestas/models.py

```
from django.db import models

class Pregunta(models.Model):
    texto_pregunta = models.CharField(max_length=200)
    fecha = models.DateTimeField('Fecha de publicación')

class Opcion(models.Model):
    pregunta = models.ForeignKey(Pregunta)
    texto_opcion = models.CharField(max_length=200)
    votos = models.IntegerField(default=0)
```

El código es sencillo de entender. Cada modelo es representado por una clase, que a su vez es una subclase de django.db.models.Model. Cada modelo tiene un número de variables, que representan un campo en la base de datos del modelo.

Cada campo está representado por una instancia de una clase llamada Field – por ejemplo la clase CharField se utiliza para guardar caracteres y la clase DateTimeField para datos del tipo fecha. Esta es la forma en que le decimos a Django qué tipo de datos contiene cada campo.

El nombre de cada instancia de models.Field (por ejemplo pregunta o fecha) es el nombre del campo, en un formato amigable para la base de datos. Este valor será usado en el código Python y la base de datos lo usará como el nombre de la columna correspondiente.

Observa que es posible usar como primer argumento de un campo (Field) un nombre más legible (entre comillas). Esto se usa vía introspección en algunas partes de Django, y sirve como documentación. Si no le asignas un nombre a este campo, Django usará el nombre del campo tomado del nombre asignado al campo en la clase. En este ejemplo, sólo hemos definido un nombre especial para Pregunta.fecha. Para los otros campos el nombre es lo suficientemente claro.

Algunas clases Field requieren de argumentos obligatorios. Por ejemplo, CharField requiere que se le pase un atributo max_length. Esto se usa no sólo en relación al esquema de la base de datos, sino también a la hora de hacer validaciones, como veremos más adelantes.

La clase Field, posee algunos argumentos opcionales; en este caso hemos fijado el atributo default a un valor de votos es igual a 0.

Finalmente, observa que se hemos definido una relación foránea usando una clase ForeignKey. Esto le informa a Django que cada Opción está relacionada con una sola Pregunta y una Pregunta puede tener múltiples opciones. Django soporta todas las relaciones de base de datos típicas como: muchos a uno, muchos a muchos y uno a uno.

Activa los modelos

Con sólo la porción de código de modelo mostrada, Django obtiene mucha información. Por ejemplo, ahora Django es capaz de:

- Crear un esquema de la base de datos (la declaración **`CREATE TABLE`**) para nuestra aplicación.
- Crear una API de acceso de datos en python para acceder a los objetos de la encuesta (Pregunta y Opción).

Pero primero necesitamos decirle a nuestro proyecto que la aplicación encuestas está instalada.

Filosofía

Las aplicaciones Django son “reusables”: Es posible usar una aplicación en múltiples proyectos, y es posible distribuir aplicaciones, debido a que no están atadas a una instalación Django en particular.

Edita el archivo **`settings.py`** otra vez y cambia la variable *INSTALLED_APPS* para incluir la cadena "encuestas" (el nombre de la aplicación), así:

mysite/settings.py

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'encuestas',
)
```

Ahora Django sabe que debe incluir la aplicación `encuestas`. Ejecutemos otro comando:

```
$ python manage.py makemigrations encuestas
```

Deberías ver algo parecido a lo siguiente:

```
Migrations for 'encuestas':
  0001_initial.py:
   -   Create model Opcion
   -   Create model Pregunta
   -   Add field pregunta to opción
```

Al ejecutar el comando makemigrations, le estamos diciendo a Django que hemos hecho algunos cambios a nuestros modelos (en este caso hemos creado unos nuevos) y que nos gustaría que los cambios sean guardados en una migración.

Las migraciones son la forma en que Django almacena los cambios de nuestros modelos (y el esquema de la base de datos) - son simplemente archivos en el disco. Puedes leer las migraciones de tu modelo si quieres; estas están en un archivo de tu proyecto en: encuestas/migrations/0001_initial.py. No te preocupes, no es necesario que las leas cada vez que Django hace algún cambio en el esquema de la base de datos, sin embargo están diseñadas para que puedan leerse y editarse, en caso de que quieras manualmente modificarlas.

Hay un comando que ejecuta las migraciones por ti y maneja el esquema de la base de datos automáticamente – este se llama migrate y lo veremos en acción un poco más adelante –pero primero veamos el SQL que las migraciones ejecutaran. El comando sqlmigrate toma el nombre de las migraciones y devuelve su respectivo SQL.

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

Deberías ver algo parecido a lo siguiente (hemos reformateado el código para legibilidad):

```
BEGIN;
CREATE TABLE "encuestas_opcion" (
       "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
       "texto_opcion" varchar(200) NOT NULL,
       "votos" integer NOT NULL);

CREATE TABLE "encuestas_pregunta" (
       "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
       "texto_pregunta" varchar(200) NOT NULL,
       "fecha" datetime NOT NULL);

CREATE TABLE "encuestas_opcion__new" (
       "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
       "texto_opcion" varchar(200) NOT NULL,
       "votos" integer NOT NULL,
       "pregunta_id" integer NOT NULL REFERENCES
           "encuestas_pregunta" ("id"));

INSERT INTO "encuestas_opcion__new" (
       "texto_opcion", "id", "votos")

SELECT "texto_opcion", "id", "votos" FROM "encuestas_opcion";

DROP TABLE "encuestas_opcion";
```

```
ALTER TABLE "encuestas_opcion__new" RENAME TO
    "encuestas_opcion";

CREATE INDEX "encuestas_opcion_5e7715cc" ON "encuestas_opcion"
    ("pregunta_id");

COMMIT;
```

Observa lo siguiente:

- La salida varía dependiendo de la base de datos que estés utilizando. (Esta salida corresponde a sqlite3).
- Los nombres de las tablas son generadas automáticamente combinando el nombre de la aplicación (encuestas) y el nombre en minúsculas del modelo – `pregunta` y `opción`. (Este comportamiento puede ser cambiado).
- Las claves primarias (IDs) se agregan automáticamente (también es posible cambiar este comportamiento).
- Por convención, Django agrega un "_id" al campo de la clave foránea. Sí, sí, también puedes sobre-escribir esto.
- La clave de relación foránea (foreign key) es hecha explicita mediante una restricción FOREIGN KEY. (Si usas PostgreSQL, no te preocupes acerca de la parte DEFERRABLE, esta es la forma en que le decimos a PostgreSQL que no implemente la clave foránea hasta el final de la transacción.)
- El SQL generado depende de la base de datos que se está usando, de manera que los tipos de campos como auto_increment (MySQL), serial (PostgreSQL), o integer primary key autoincrement (SQLite) son manejadas por ti automáticamente. Lo mismo va para el uso de comillas en los nombres de campos – por ejemplo, el uso de comillas simples o dobles.
- El comando sqlmigrate no ejecuta automáticamente el SQL de la base de datos –sólo muestra en pantalla la salida, para que veas lo que django piensa que se requiere. Es útil para comprobar lo que Django va a hacer o si tienes administradores de base de datos que requieren de scripts para realizar cambios. Si lo quieres, puedes copiar y pegar este SQL en la interfaz de tu base de datos. No obstante, como veremos pronto, Django provee una forma más sencilla de ingresar SQL a la base de datos.

Si estás interesado puedes ejecutar python manage.py check, este comando comprueba cualquier problema que pueda haber en tu proyecto, sin hacer cambios en las migraciones, ni tocar la base de datos

Ahora, ejecuta el comando `migrate` para crear las tablas del modelo en tu base de datos:

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

El cual muestra una salida como la siguiente:

```
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles,
  Apply all migrations: auth, contenttypes, encuestas, sessions,
admin
Synchronizing apps without migrations:
  Creating tables...
  Installing custom SQL...
  Installing indexes...
Installed 0 object(s) from 0 fixture(s)
Running migrations:
  Applying encuestas.0001_initial... OK
```

El comando migrate toma todas las migraciones que no han sido aplicadas (Django rastrea cada una de las migraciones aplicadas a los modelos, usando una tabla especial llamada django_migrations) y ejecuta el sql de 'sqlmigrate' en tu base de datos para todas las aplicaciones en *INSTALLED_APPS* que no existían ya en la base de datos. Esto crea todas las tablas, los datos iníciales y los índices para cada aplicación que se haya agregado al proyecto desde la última vez que se ejecutó migrate - esencialmente, sincronizando los cambios hechos a tus modelos con el esquema de la base de datos. Puedes ejecutar el comando migrate tan frecuentemente como desees, ya que sólo creará tablas que no existan.

Las migraciones son muy poderosas y te permiten cambiar tus modelos a lo largo del tiempo, mientras desarrollas tu proyecto, sin que sea necesario borrar la base de datos, tablas o crear una nueva - su especialidad es actualizar tu base de datos en funcionamiento, sin perder datos. Cubriremos esto más adelante en profundidad, por ahora solo recuerda seguir estos tres pasos para hacer cambios en tus modelos:

- Cambia tu modelo (en models.py)
- Ejecuta python manage.py makemigrations para crear las migraciones para los cambios
- Ejecuta python manage.py migrate para aplicar esos cambios a la base de datos.

La razón de separar los comandos para hacer y aplicar las migraciones es debido a que puedes aplicar y guardar las migraciones en una versión de control y luego enviarlas junto con tu aplicación. Esto no sólo simplifica el desarrollo, sino también es útil a otros desarrolladores y en producción.

Algunos otros comandos útiles que puedes usar:

- python manage.py check – Busca errores en la construcción de tus modelos.
- python manage.py sqlcustom encuestas – Muestra las sentencias SQL definidas manualmente para la aplicación (por ejemplo, modificaciones de tablas y restricciones).

- python manage.py sqlclear encuestas Muestra las sentencias DROP TABLE necesarias para esta aplicación, de acuerdo a las tablas que ya existen en la base de datos, si corresponde.

- python manage.py sqlindexes encuestas –Muestra la salida de las sentencias CREATE INDEX para esta aplicación.

- python manage.py sqlall encuestas – Una combinación de todas las sentencias SQL de los comandos *sql*, *sqlcustom*, y *sqlindexes*.

Revisar la salida de esos comandos pueden ayudarte a comprender lo que realmente ocurre tras bambalinas.

Jugando con la API

Ahora, usemos el intérprete interactivo de Python para jugar con la API que Django ofrece. Para invocar el intérprete interactivo, usamos este comando:

```
$ python manage.py shell
```

Usamos este comando en vez de teclear simplemente "python", debido a que manage.py configura el entorno del proyecto por nosotros, importa el archivo *settings.py* de manera automática.

Sin usar manage.py:

Si por alguna razón no quieres usar el comando manage.py no hay problema, solo necesitas importar las variables de entorno de la variable DJANGO_SETTINGS_MODULE de el archivo misitio.settings y ejecutar python en el mismo directorio donde está el archivo manage.py (siempre que el directorio este en la ruta de búsqueda de Python)

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

Si el segundo comando lanza un error del tipo `AttributeError`, probablemente estas usando una versión que no corresponde con la de este tutorial.

Una vez que estés en el shell interactivo, puedes explorar la API de base de datos:

```
# Primero importamos los modelos que hemos creado
 >>> from encuestas.models import Pregunta, Opcion

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

# Creamos una nueva pregunta.
```

```
# El soporte para zonas horario está configurado en el fichero
# de configuración de forma predeterminada, así que Django
# espera un valor datetime con tzinfo para el campo fecha.
# Usamos timezone.now() en lugar de datetime.datetime.now (),
# Esto está bien por ahora.
>>> from django.utils import timezone
>>> p= Pregunta(texto_pregunta="¿Qué cuentas?",
       fecha = timezone.now())

# Guardamos el objeto en la base de datos. Llamando al método
# save() explícitamente.
>>> p.save()

# Ahora tenemos un identificador ID. Observa que podemos obtener
# “1L" en vez de "1", dependiendo de la base de datos que
# estés utilizando. Esto no es importante; solo significa que
# la base de datos prefiere usar números enteros como números
# enteros largos de python.
>>> p.id
1
# Accedemos a una columna a través de sus atributos python.
>>> p.texto_pregunta
"¿Qué cuentas?"
>>> p.fecha
datetime.datetime(2015, 1, 19, 23, 30, 44, 775217, tzinfo=<UTC>)

# Cambiamos los valores, cambiando los atributos y llamando a él
# método guardar: save().
>>> p.texto_pregunta = "¿Qué pasa?"
>>> p.save()
# objects.all() Muestra todas las encuestas de la base de datos.
>>> Pregunta.objects.all ()
[<Pregunta: Pregunta object>]
```

Espera, espera un minuto. < Pregunta: Pregunta object> es una representación completamente inútil de este objeto. Reparemos esto editando el modelo encuestas (en el archivo encuestas/models.py) agregando un método __str__() tanto a Pregunta como a Opción así:

encuestas/models.py

```
from django.db import models

class Pregunta(models.Model):
    # ... Aquí van los campos

    def __str__(self):           # __unicode__ en Python 2
        return self.Pregunta

class Opcion(models.Model):
    # ... Aquí van los campos

    def __str__(self):            # __unicode__ en Python 2
        return self.Opcion
```

Es importante agregar métodos __str__() a tus modelos, no sólo para tu propia legibilidad cuando trabajes con el intérprete interactivo, sino también porque la representación de los objetos se usa en la interfaz de administración que Django genera automáticamente (como la vista del índice de listado, que veremos más adelante).

¿ Métodos __str__() o __unicode__()?

En Python 3 es fácil, solo usa métodos __str__.

En python 2, es necesario definir métodos __unicode__, que devuelvan valores Unicode en su lugar. Django usan Unicode de forma predeterminada. Todos los datos guardados en la base de datos se convierten a Unicode cuando son devueltos.

Los modelos de Django tienen un valor por defecto __str__() este método llama a __unicode__() y convierte el resultado a UTF-8 bytestring. Esto significa que el método unicode(p) retornara cadenas Unicode, y el método str (p) devolverá una cadena normal, con los caracteres codificado como UTF-8.

Si todo esto no tiene sentido para ti, solo usa Python 3 y todo funcionara bien.

Observa que estos son métodos normales de Python. Agreguemos un método personalizado más, solo para demostración:

encuestas/models.py

```
import datetime

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

class Pregunta(models.Model):
    texto_pregunta = models.CharField(max_length=200)
    fecha = models.DateTimeField('Fecha de publicación')

    def __str__(self):              # __unicode__ en Python 2
        return self.Pregunta

    def publicado_recientemente(self):
        return self.fecha >= timezone.now()
             -datetime.timedelta(days=1)
```

Observa que agregamos datetime y from django.utils import timezone para referenciar al método estándar de python datetime y los utilitarios de zonas horario de Django: django.utils.timezone, respectivamente.

Guardamos los cambios y comenzamos un nuevo shell interactivo python ejecutando python manage.py shell otra vez:

```
>>> from encuestas.models import Pregunta, Opcion

# Nos aseguramos que el método __str__() trabaje bien.
>>> Pregunta.objects.all()
```

```
[<Pregunta: ¿Qué pasa?>]

# Django provee una rica API para búsquedas en la base de datos
# usando argumentos clave
>>> Pregunta.objects.filter(id=1)
[<Pregunta: ¿Qué pasa?>]
>>> Pregunta.objects.filter(texto_pregunta__startswith='¿Qué')
[<Pregunta: ¿Qué pasa?>]

# Obtenemos una pregunta que fue publicada en este año.
>>> from django.utils import timezone
>>> año_actual = timezone.now().year # Usando python3
>>> Pregunta.objects.get(fecha__year=año_actual)
[<Pregunta: ¿Qué pasa?>]

# Directamente usando un filtro.
>>> Pregunta.objects.get(fecha__year=2015)
[<Pregunta: ¿Qué pasa?>]


# Instanciar un ID que no exista, lanzara una excepción.
>>> Pregunta.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Pregunta matching query does not exist.

# Las búsquedas por clave primaria, son las opciones más comunes
# para lo cual django, provee atajos para búsquedas exactas
# por clave primaria.
# Lo siguiente es idéntico a Pregunta.objects.get(id=1).
>>> Pregunta.objects.get(pk=1)
[<Pregunta: ¿Qué pasa?>]

# Ahora veamos si nuestro método personalizado trabaja.
>>> p = Pregunta.objects.get(pk=1)
>>> p.publicado_recientemente()
True

# Démosle a Preguntas un par de Opciones. Para ello llamamos al
# constructor  create, para crear un nuevo objeto Opción, no es
# una declaración INSERT, solo agrega una Opción al conjunto de
# opciones disponibles y devuelve un nuevo objeto Opción.
# Django crea un conjunto para mostrar la “otra cara” de una
# relación ForeignKey  (por ejemplo: preguntas a través de las
# opciones)Las cuales pueden ser accedidas a través de la API.
>>> p = Pregunta.objects.get(pk=1)

# Muestra cualquier Opción del conjunto de relaciones. -ninguno
# hasta ahora
>>> p.opcion_set.all()
[]

# Creamos tres Opciones.
```

```
>>> p.opcion_set.create(texto_opcion = 'No mucho', votos=0)
<Opcion: No mucho>
>>> p.opcion_set.create(texto_opcion = 'Nada', votos=0)
<Opcion: Nada>
>>>c = p.opcion_set.create(texto_opcion = 'De hacker otra vez',
       votos = 0)

# Accedemos a través de la API a los objetos Pregunta.
>>> c.pregunta
[<Pregunta: ¿Qué pasa?>]

# Y viceversa: Los objetos Pregunta tienen acceso a los objetos
# Opcion.
>>> p.opcion_set.all()
[<Opcion: No mucho>, <Opcion: Nada>, <Opcion: De hacker otra
vez>]
>>> p.Opcion_set.count()
3

# La API automáticamente sigue las relaciones tan lejos como lo
# necesites. Usa guiones bajos (__) para separar las relaciones.
# Esto funciona en varios niveles y tan profundo como quieras;
# no hay límite.
# Ahora encuentra todas las Opciones para cualquier Pregunta
# publicada en este año (rehusando la variable “año_actual” que
# creamos anteriormente)
>>> Opcion.objects.filter(pregunta__fecha__year=año_actual)
[<Opcion: No mucho>, <Opcion: Nada>, <Opcion: De hacker otra
vez>]

#Para borrar una Opción, usa el método delete() .
>>> c = p.opcion_set.filter(texto_opcion__startswith='De')
>>> c.delete()
```

Para más información sobre relaciones de modelos, consulta “Como acceder a objetos relacionados” Para más información sobre campos de búsqueda puedes consultar “búsquedas en campos” Para detalles completos sobre la API de la base de datos, considera leer “La referencia de API de bases de datos”.

¿Qué sigue?

Cuando te sientas cómodo con la API, continúa con la segunda parte de este tutorial para configurar el sitio de administración automático de Django.

CAPÍTULO 04

Escribe tu primera aplicación Django, parte 02

Continuamos con la segunda parte de este tutorial, seguimos construyendo una aplicación de encuestas Web y en esta parte, nos centraremos en explorar y modificar la interfaz administrativa de Django, generada automáticamente.

Filosofía

La Generación de sitios administrativos para que el personal o tus clientes agreguen, cambien y borren el contenido, es el trabajo aburrido que no requiere mucha creatividad. Por esta razón, Django automatiza enteramente la creación de las interfaces administrativas para los modelos.

Django fue escrito en un ambiente de redacción, con una separación muy clara entre los "editores de contenido" y el sitio "público". Los encargados del sitio utilizan el sistema para agregar nuevas noticias, eventos, marcadores, deportes, etc., y ese contenido es visualizado en el sitio público. Django soluciona el problema de crear una interfaz unificada para que administradores del sitio editen el contenido.

El área administrativa no está pensada para ser utilizado por los visitantes del sitio. Esta debe ser editada por el encargado del el sitio: el administrador.

Crea un usuario administrador

Lo primero que necesitas es crear un usuario que puede identificarse como el administrador del sitio Web, para ello ejecuta el siguiente comando:

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

Veamos la salida del comando createsuperuser interactivamente:

Introduce un nombre de usuario y presiona enter:

```
Username: admin
```

También te pedirá una dirección de correo electrónico:

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

Como paso final introduce una contraseña. El intérprete interactivo te pedirá introducir dos veces la misma contraseña, la segunda vez solo como confirmación de la primera.

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

Inicia el servidor del desarrollo

El sitio de administración de Django se activa de forma predeterminada – aunque es opcional.

Inicia el servidor del desarrollo y explora el sitio de administración.

Recuerdas en el Tutorial 1, como iniciamos el servidor de desarrollo así:

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

Ahora, abre un navegador web y dirígete a "/admin/" en tu dominio local – por ejemplo: http://127.0.0.1:8000/admin/. Deberías ver una pantalla de identificación como esta:

Imagen 2.1 *Pagina de identificación de Django*

¿No es lo mismo que ves?

En este punto, si en lugar de ver la pagina de identificación, ves una página de error tal como esta:

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

Entonces probablemente estas usando una versión de Django que no corresponde con la versión de este tutorial.

Cambiar el idioma predeterminado

El archivo settings.py contiene una variable que te permite especificar el idioma a usar, puedes encontrar la lista completa de idiomas en: http://www.i18nguy.com/unicode/language-identifiers.html:

```
LANGUAGE_CODE = 'es'
```

Entra al sitio de administración

Ahora, tras registrarte en la página de identificación, que creamos en el paso anterior, puedes ver una página de índice en la interfaz administrativa así:

Imagen 2.2 *Pagina de índice de la interfaz administrativa*

Deberías poder ver algunos tipos de contenido editable, que incluye Grupos y Usuarios. Estos son proporcionados por el framework de autentificación, incluidos en Django en el paquete django.contrib.auth.

Hacer que nuestra aplicación de encuestas se pueda editar en el admin

¿Pero dónde está nuestra aplicación de encuestas? No se visualiza en la página del índice de la interfaz administrativa.

Justamente tenemos que hacer una cosa más: necesitamos decirle a la interfaz administrativa de Django que registre nuestro objeto Pregunta de nuestra aplicación Encuestas, para hacer esto edita el fichero admin.py en el directorio encuestas así:

encuestas/admin.py

```
from django.contrib import admin
from encuestas.models import Pregunta

admin.site.register(Pregunta)
```

No es necesario reiniciar el servidor del desarrollo para ver los cambios en acción. Normalmente, el servidor auto-recarga el código cada vez que modificamos un fichero, solamente la acción de crear un nuevo fichero no acciona la lógica de auto-recarga.

Explora libremente las funcionalidades de la interfaz administrativa

Ahora que hemos registrado el modelo `Pregunta` de nuestra aplicación `Encuestas`, Django sabe como mostrarla en la página de índice de la interfaz administrativa:

Imagen 2.3 *Pagina de índice; registrando un modelo*

Da clic en "Preguntas", para ver la página de preguntas, ahora estas en la "lista de cambios". Esta página muestra todas las preguntas en la base de datos y te permite cambiar y agregar preguntas. Recuerdas la pregunta "¿Qué pasa?" que creamos en el tutorial de la primera parte usando la terminal, aquí esta:

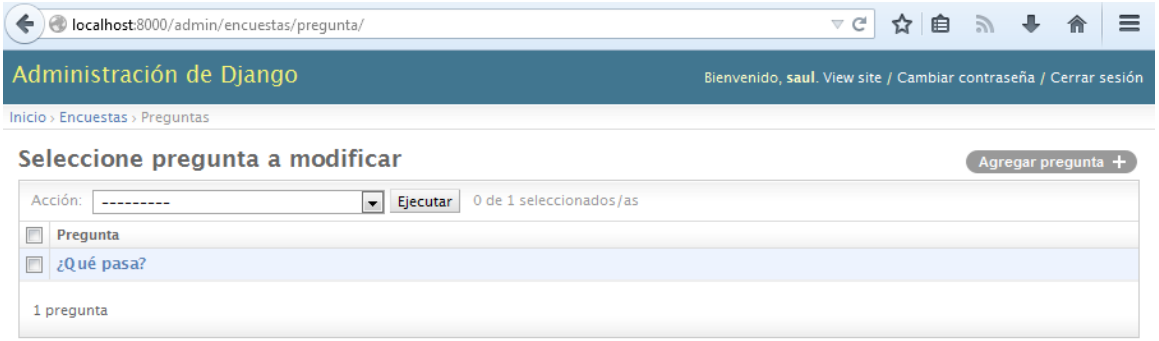


Imagen 2.4 *Pagina de lista de cambios del formulario preguntas*

Da clic en la pregunta "¿Qué pasa?" para editarla, así:

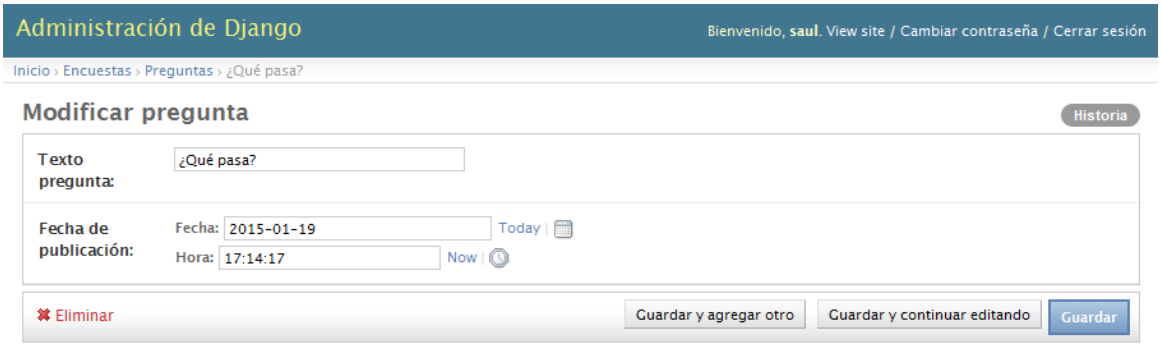


Imagen 2.5 *Formulario para editar preguntas*

Alguna cosas a observar aquí:

- El formulario es automáticamente generado para el modelo Pregunta.
- Los diversos tipos de campo del modelo (DateTimeField, CharField) corresponden a un apropiado widget HTML de entrada de datos. Cada tipo de campo sabe visualizarse a sí mismo, en la interfaz administrativa de Django.

- Cada campo DateTimeField proporciona accesos directos a JavaScript, (widgets). Las fechas proporcionan un enlace a un widgets de tipo calendario "Hoy" y la fecha "Ahora" muestra el icono de un reloj, con una apropiada lista de opciones para cambiar la hora, estos son atajos comunes y están incorporados en la interfaz administrativa.

En la parte inferior de la página, existen cuatros botones:

- **Guardar** – Guarda los cambios y vuelve a la página de lista de cambios para este tipo de objeto.
- **Guardar y continuar editando** – Guarda los cambios al objeto y recarga la página del admin para continuar editando.
- **Guardar y agregar otro** – Guarda los cambios y carga una nueva página en blanco para añadir una nuevo objeto.
- **Eliminar** – Muestra una página de confirmación para borrar un objeto.

Si el valor de la "fecha" no corresponde con el tiempo en que se creó la pregunta en el tutorial 1, probablemente olvidaste agregar el valor correcto al archivo setting.py a la variable TIME_ZONE. Cámbialo, recarga la página y checa que los valores correctos aparezcan ahora.

Cambia la fecha dando clic en "hoy" y "ahora". Da clic en "Guardar y continuar editando" Da clic en "historia" en la parte superior derecha y podrás ver una lista de los cambios hechos a los objetos a través de la interfaz administrativa, con la marca de tiempo y el usuario que realizó los cambios:

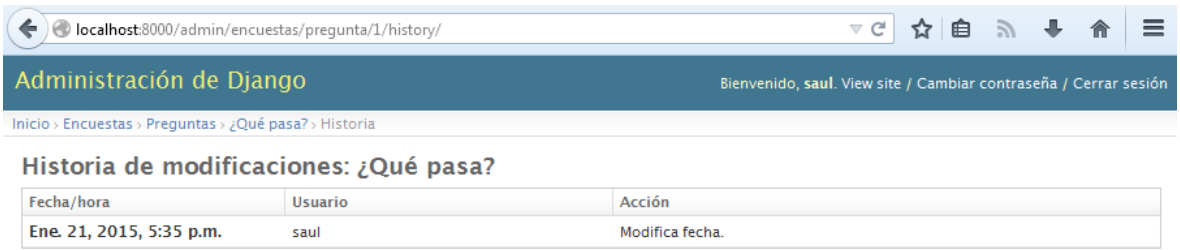


Imagen 2.6 *Pagina de historia de un objeto*

Personaliza la Interfaz administrativa

Tomate algunos minutos para admirar todo el código *que no* tuvimos que escribir. Al registrar el modelo Pregunta con admin.site.register (Encuesta), Django pudo construir una representación predeterminada para el formulario Pregunta.

A menudo, querrás personalizar la apariencia y la forma en que trabaja la interfaz administrativa. Puedes hacerlo diciéndole a Django las opciones que quieres, cuando registres el objeto en cuestión.

Veamos cómo trabaja, reordenando los campos del formulario de acuerdo a su fecha de publicación, modifica el archivo admin.py, reemplazando la línea de admin.site.register (Encuesta) con:

encuestas/admin.py
```
from django.contrib import admin
from encuestas.models import Pregunta

class Preguntadmin(admin.ModelAdmin):
    fields = ['fecha', 'texto_pregunta']

admin.site.register(Pregunta, Preguntadmin)
```

Hemos seguido un patrón – creamos un objeto para un modelo en la interfaz administrativa, después lo pasamos como el segundo argumento a admin.site.register() – por lo que en cualquier momento puedes necesitar cambiar las opciones del objeto en la interfaz administrativa.

Por ejemplo, este cambio en particular hace que el campo “Fecha” aparezca antes del campo “Pregunta” así:

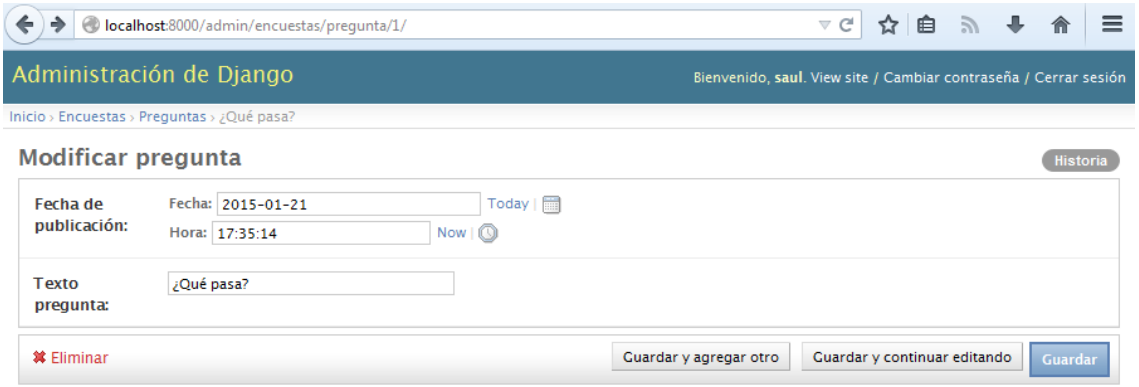


Imagen 2.7 *Reordenando campos en un formulario*

Esto no es impresionante con sólo dos campos, pero con formulario con docenas de campos en la interfaz administrativa, escoger un orden intuitivo es un detalle importante de usabilidad.

Y hablando de formularios con docenas de campos, también podemos dividir los formularios en un conjunto de campos, para lograr mayor visibilidad usando la clase fieldsets, volvamos a modificar admin.py así:

encuestas/admin.py
```
from django.contrib import admin
from encuestas.models import Pregunta

class PreguntaAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['texto_pregunta']}),
        ('Información de la fecha', {'fields': ['fecha']}),
    ]

admin.site.register(Pregunta, PreguntaAdmin)
```

El primer elemento de cada tupla en fieldsets es el titulo de cada campo, para cada conjunto de campos, aquí está ahora el formulario:

Imagen 2.8 *Reordenando campos usando fieldsets*

También puedes asignar clases arbitrarias de HTML a cada fieldset. Django proporciona una clase "collapse" (colapsar) que muestra un fieldset en particular inicialmente colapsado. Esto es útil cuando tienes un formulario que contienen un número de campos que no son muy usados en general:

encuestas/admin.py

```
from encuestas.models import Pregunta

class PreguntaAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['texto_pregunta']}),
        (' Información de la fecha', {'fields': ['fecha'],
            'classes': ['collapse']}),
    ]

admin.site.register(Pregunta, PreguntaAdmin)
```

Solo tienes que darle clic a “mostrar” para expandir los fieldsets colapsados y viceversa puedes dar clic en “ocultar” para colapsar el grupo de campos, así:

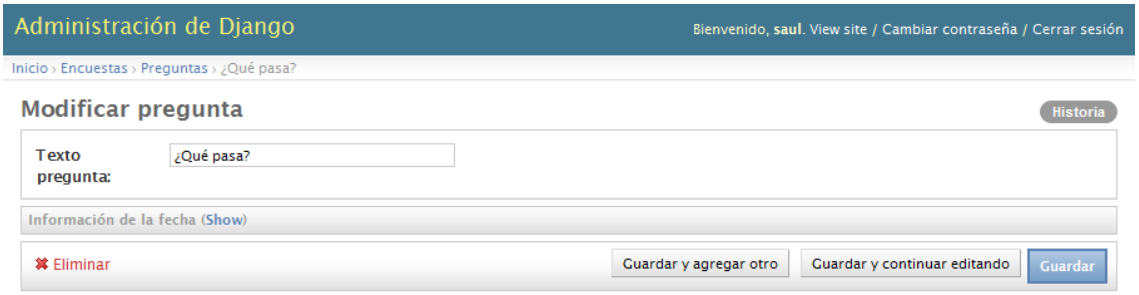


Imagen 2.9 *Expandiendo y colapsando campos*

Agregando relaciones a objetos

Bien, ya tenemos nuestra página de administración para las preguntas. Pero las preguntas tienes múltiples opciones y nuestra página de administración no muestra las opciones.

Hay dos maneras de solucionar este problema. El primero es registrar el modelo Opción en la interfaz administrativa como hicimos con Pregunta. Esto es sencillo.

encuestas/admin.py

```
from django.contrib import admin
from encuestas.models import Opcion, Pregunta
# ...

admin.site.register(Opcion)
```

Ahora tenemos tanto a Preguntas como a Opcions registradas,

Imagen 2.10 *Registrando el modelo Opción*

Espera, espera un minuto, Opcions no es el nombre en plural de Opción ¿Que ha pasado? Bueno de forma predeterminada Django construye el plural de un objeto agregando una "s" al final del nombre de la clase de un modelo, que en casos como este no siempre es lo más adecuado, sin embargo hay una forma muy sencilla de corregir esto. Agrega una clase Meta interna y usa el nombre en plural del objeto con verbose_name_plural:

encuestas/models.py

```
class Opcion(models.Model):
    pregunta = models.ForeignKey(Pregunta)
    texto_opcion = models.CharField(max_length=200)
    votos = models.IntegerField(default=0)

    class Meta:
        verbose_name = 'opción'# El nombre del modelo
        verbose_name_plural = 'opciones'# El nombre en plural

    def __str__(self):              # __unicode__ en Python 2
        return self.Opcion
```

Imagen 2.11 *Usando nombres en plural con verbose_name_plural*

Observa que también agregamos una variable verbose_name, esta le dice a Django: usa este nombre de forma predeterminada en la interfaz administrativa (con todo y acento).

Ahora "Opción" está disponible en la interfaz de administración. El formulario para "Agregar una opción" se parece a este:

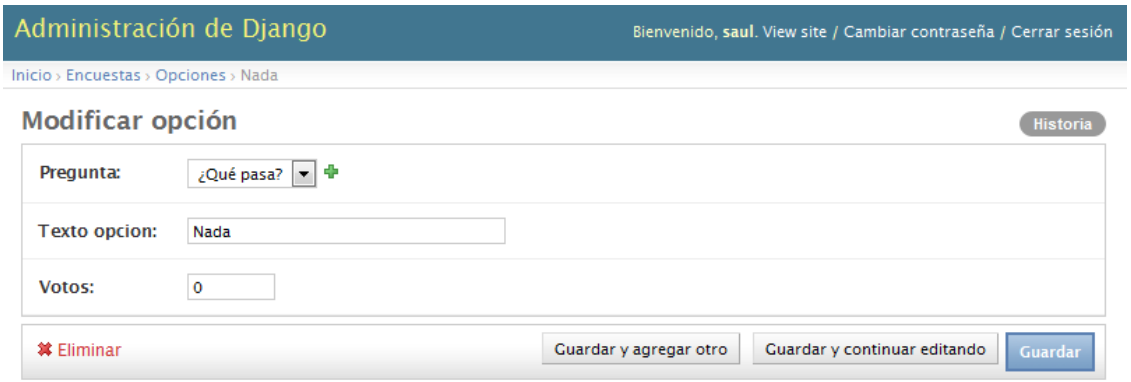


Imagen 2.12 *Ejemplo de un formulario para agregar opciones*

En este formulario, el campo "Pregunta" es una caja de selección que contiene todas las preguntas de nuestra base de datos. Django sabe que los campos ForeignKey deben ser representados en la interfaz administrativa como una caja de selección. En nuestro caso, solamente tenemos una pregunta, en este punto.

También observa que Pregunta contiene un enlace "Agregar otra" (un signo de + en color verde) para agregar otra pregunta. Cada objeto con un campo ForeignKey contiene una relación a otro y esto es gratis. Cuando das clic en "Agregar otra" obtienes acceso a una ventana emergente que contiene un nuevo formulario para agregar una nueva pregunta y "guardarla" en la base de datos, lo que te permite dinámicamente agregar una nueva pregunta o seleccionar la que estas mirando.

Pero realmente, ésta es una manera muy ineficaz de agregar los objetos Opción al sistema, no sería mejor si pudiéramos agregar un grupo de opciones directamente al crear el objeto Pregunta, veamos cómo hacerlo.

Primero quita la llamada al método register() para el modelo Opción y edita el código del registro de Pregunta así:

encuestas/admin.py

```
from django.contrib import admin
from encuestas.models import Opcion, Pregunta

class OpcionInline(admin.StackedInline):
    model = Opcion
    extra = 3

class PreguntaAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['texto_pregunta']}),
        ('fecha', {'fields': ['fecha'],
            'classes': ['collapse']}),
    ]
    inlines = [OpcionInline]

admin.site.register(Pregunta, PreguntaAdmin)
```

Esto le dice a Django: Los objetos "Opcion" son editados en la página de administración de Preguntas. En este caso editamos el valor predeterminado y proporcionamos 3 campos extras para cada "Opción".

Recarga la página "Agregar pregunta" para ver cómo luce ahora:

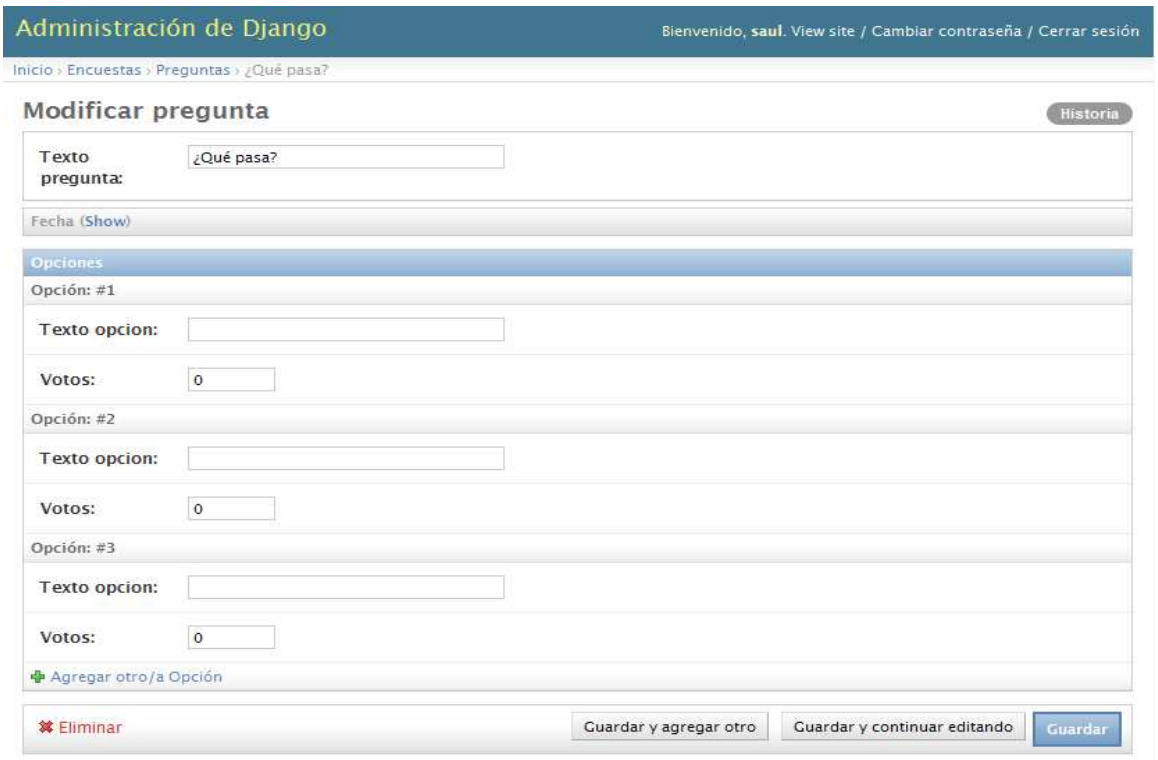


Imagen 2.13 *Ejemplo de el uso de OpcionInline*

Esto funciona así: Tenemos tres casillas para las opciones relacionadas – según lo especificado con extra – y cada vez que volvamos a la página de "cambios" podemos generar otras tres opciones adicionales.

Al final de las tres casillas encontraras un enlace que dice "Agregar otra Opción" Si das clic en el podrás agregar otra casilla. Si quieres quitar la casilla recién añadida, da

clic sobre la X en la parte superior de la casilla añadida. Toma en cuenta que no podrás quitar las tres casillas originales. Esta imagen muestra una casilla agregada:

Imagen 2.14 *Ejemplo de una casilla extra*

Tenemos un pequeño problema aquí, esto toma mucho espacio en la pantalla para mostrar todos los campos y objetos relacionados que tenemos. Por esta razón, Django ofrece una manera tabular de visualizar los objetos relacionados en línea; solo necesitas cambiar la declaración de OpcionInline y agregar TabularInline en lugar de StackedInline así:

encuestas/admin.py

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

Con la clase TabularInline (en vez de StackedInline), los objetos relacionados se visualizan en un formato más compacto, basado en un formato tipo tabla.

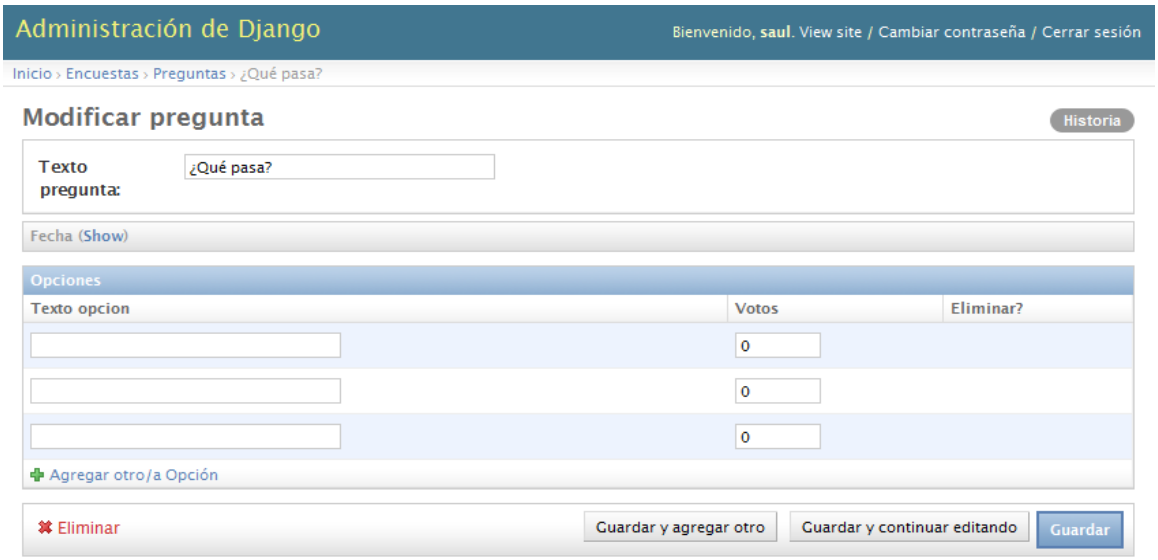


Imagen 2.15 *Ejemplo de TabularInline*

Observa que ahora tenemos una columna extra “Eliminar”, que te permite remover filas agregadas usando el botón “Agregar otra Opción” y filas que ya han sido guardadas.

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

Inicia el servidor del desarrollo

El sitio de administración de Django se activa de forma predeterminada – aunque es opcional.

Inicia el servidor del desarrollo y explora el sitio de administración.

Recuerdas en el Tutorial 1, como iniciamos el servidor de desarrollo así:

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

Ahora, abre un navegador web y dirígete a "/admin/" en tu dominio local – por ejemplo: http://127.0.0.1:8000/admin/. Deberías ver una pantalla de identificación como esta:

Imagen 2.1 *Pagina de identificación de Django*

¿No es lo mismo que ves?

En este punto, si en lugar de ver la pagina de identificación, ves una página de error tal como esta:

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

Entonces probablemente estas usando una versión de Django que no corresponde con la versión de este tutorial.

Ahora puedes dar clic en la parte superior de cada columna para ordenar por valores – excepto en el caso fue_publicado_recientemente, ya que no está soportado el ordenamiento de métodos arbitrarios. Observa que la cabecera de la columna para fue_publicado_recientemente es de forma predeterminada el nombre del método con los caracteres subrayados substituidos por espacios en blanco.

Podemos mejorar esto asignándole al método fue_publicado_recientemente (en encuestas/models.py) algunos atributos, como sigue:

encuestas/models.py

```
class Pregunta(models.Model):
    #... campos

    def fue_publicado_recientemente (self):
        return self.fecha >= timezone.now() -
            datetime.timedelta(days=1)

    fue_publicado_recientemente.admin_order_field = 'fecha'
    fue_publicado_recientemente.boolean = True
    fue_publicado_recientemente.short_description =
        '¿Es reciente?'
```

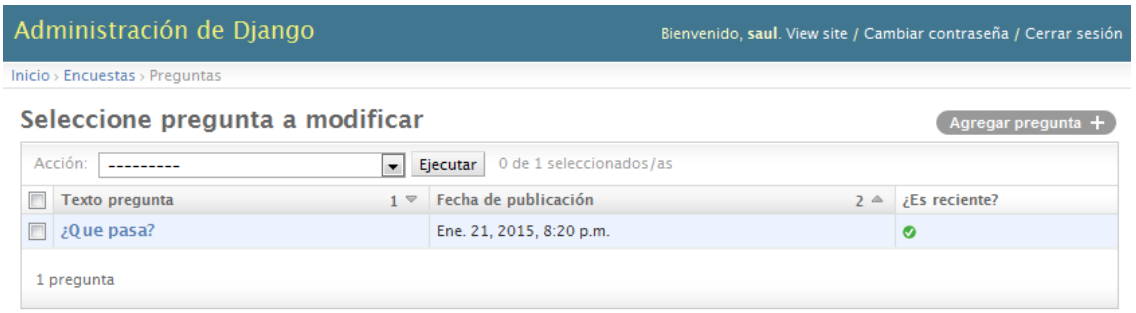


Imagen 2.18 *Pagina de lista de cambios usando atributos.*

Ahora edita el archivo encuestas/admin.py de nuevo, para agregar otra mejora a la página de la lista de cambios, un filtro; usando `list_filter`. Agrega la línea siguiente a `PreguntaAdmin`:

```
list_filter = ['fecha']
```

Esto agrega un "filtro" a la barra lateral que permite a las personas filtrar la lista de cambios por fecha, usando en este caso las fechas de las preguntas publicadas:

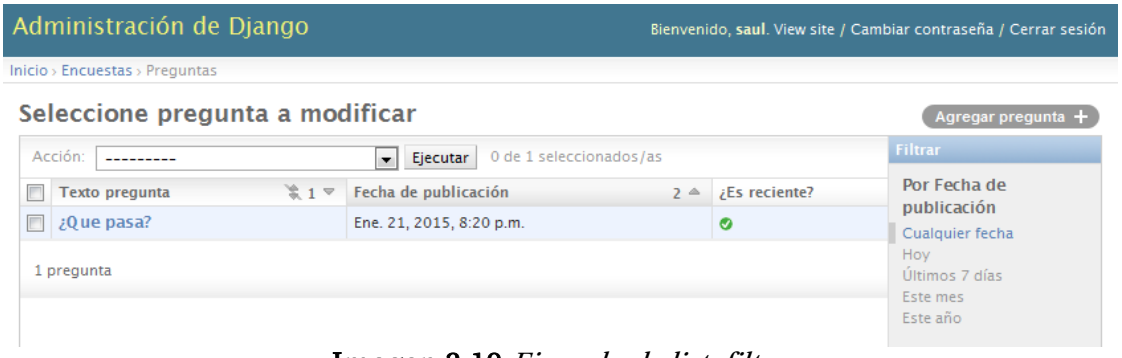


Imagen 2.19 *Ejemplo de list_filter*

El tipo de filtro a mostrar depende del tipo de campo que estés filtrando Ya que el campo fecha es un tipo DateTimeField, Django sabe como mostrar las opciones apropiadas para dicho filtro: “Cualquier fecha,” “hoy”, “Últimos 7 días”, “este mes”, “este año.”

Esto ya va tomando forma. Con la siguiente línea agregamos capacidad de búsqueda:

```
search_fields = ["texto_pregunta"]
```

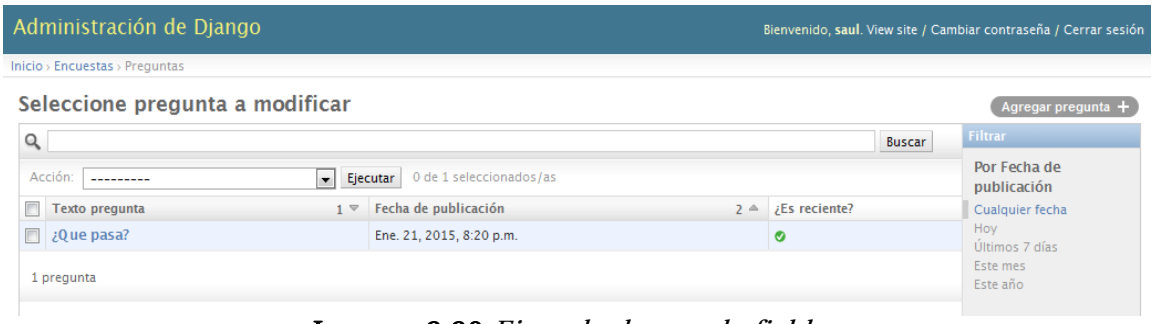


Imagen 2.20 *Ejemplo de search_fields*

Esto agrega una caja de búsqueda en la parte superior de la lista de cambios. Cuando alguien entra buscando algún término o palabra, Django busca la palabra en el campo texto_pregunta. Puedes utilizar tantos campos como quieras – aunque utiliza LIKE detrás de escena para recuperar la información de la base de datos, lo más razonable es limitar el número de campos para realizar búsquedas, un número pequeño hace más fácil las búsquedas en la base de datos.

Finalmente, porque el objeto Pregunta maneja fechas, sería conveniente poder filtrar y ordenar las fechas de forma jerárquica. Agrega esta línea:

```
date_hierarchy = "fecha"
```

Eso agrega a la navegación una jerarquía basada en fechas, arriba de la página de la lista de cambios, en la parte superior se mostraran los años y después mostrara los meses y, en última instancia, los días en que se hayan publicado preguntas.

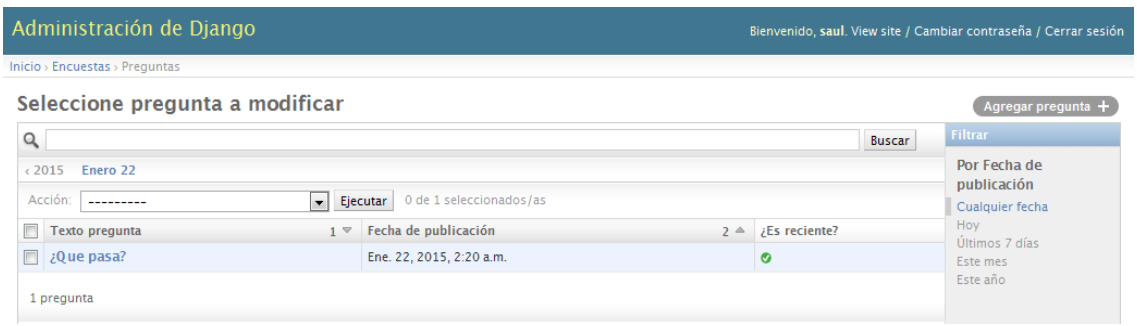


Imagen 2.21 *Ejemplo de date_hierarchy*

Ahora es un buen momento para observar todos los cambios que hemos hecho a la página de lista de cambios de forma sencilla: le hemos agregado búsquedas, filtros, navegación jerárquica, hemos ordenado las columnas y todo trabaja bien en conjunto, como lo hubieses pensado. Una última cosa la página de lista de cambios

muestra de forma predeterminada 100 objetos por cada página, antes de agregar la paginación automática.

Modifica la apariencia de la interfaz administrativa

Claramente, tener al inicio de cada página la leyenda “Administración de Django” es ridículo. Este es solo un lugar para colocar texto.

Esto es sencillo de cambiar, usando el sistema de plantillas de Django. La interfaz administrativa esta creada con Django en sí mismo y la interfaz usan el propio sistema de plantillas

Personalizar las plantillas de tu proyecto

Crea un directorio de plantillas en el directorio de tu proyecto, por convención llámalo “templates”. Las plantillas pueden alojarse en cualquier lugar del sistema de archivos al que Django tenga acceso. (Django se ejecuta sin importar el usuario que ejecuta el servidor) Sin embargo, conservar las plantillas dentro del proyecto es una buena convención para seguir.

Abre el archivo de configuración (en misitio/settings.py, recuerdas) y agrega la variable TEMPLATE_DIRS:

```
mysite/settings.py
TEMPLATE_DIRS = [os.path.join(BASE_DIR, 'templates')]
```

TEMPLATE_DIRS es un iterable de directorios del sistema de ficheros que se encarga de controlar y cargar las plantillas de Django. Es el camino de búsqueda de las plantillas de Django.

Ahora crea un directorio llamado `admin` dentro de `templates` y copia la plantilla admin/base_site.html del directorio del código fuente de Django mismo, que se encuentra en el directorio (django/contrib/admin/templates).

¿Donde están los archivos fuente de Django?

Si tienes dificultades al encontrar los archivos fuente de Django, el lugar en el que están localizados en tu sistema, puedes usar el siguiente comando para encontrar la ruta en que se encuentran en tu sistema:

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

Después, solo tienes que editar y remplazar:

```
{{ site_title|default:_('Django administration') }}
```

Y substituir el texto genérico de Django por el nombre de tu sitio o como mejor te parezca, también puedes sustituir otras líneas así:

```
<h1 id="site-name"><a href="{% url 'admin:index' %} ">
        Administración de Encuestas</a></h1>
```

Esta Plantilla contiene porciones de texto como `{% block branding %}` y `{{ title }}`. Las {% y {{ son etiquetas que son parte de el lenguaje de plantillas de Django. Cuando Django renderiza admin/base_site.html estas plantillas son evaluadas para producir la pagina final en HTML. No te preocupes si no tienen sentido para ti ahora – ahondaremos mas sobre el tema en la *parte 3 de este tutorial*.

Observa que cualquiera de las plantillas de la interfaz administrativa de forma predeterminada puede ser reemplazada o sobrescrita, para sobre-escribir una plantilla debes hacer la misma cosa que hicimos con base_site.html –copiarla al directorio predeterminado o en el directorio de tus plantillas y hacer los cambios pertinentes.

Personalizar las plantillas de tus aplicaciones

Algún astuto lector puede preguntarse: Pero si la configuración de TEMPLATE_DIRS estaba vacía por default ¿cómo es que Django encontraba las plantillas de la interfaz administrativa? La respuesta es simple, de forma predeterminada, Django busca automáticamente un sub-directorio templates/ dentro de cada subdirectorio de cada aplicación para usar las plantillas. (No olvides que django.contrib.admin es una aplicación)

Nuestra aplicación de Encuestas no es muy compleja y no necesita de plantillas personalizadas en la interfaz administrativa. Sin embargo si requiriéramos de algo más sofisticado y que necesitara de modificar las plantillas estándar de Django para agregar alguna funcionalidad, sería más sencillo modificar las plantillas de la aplicación, en lugar de modificar las del propio proyecto. De esta manera podríamos incluir la aplicación de Encuestas en un nuevo proyecto y nos aseguraríamos de encontrar las plantillas personalizadas que necesitara la aplicación.

Personalizar la pagina de índice del admin

De igual forma, tal vez quieras modificar la apariencia y el diseño de Django para la página de índice de la interfaz administrativa.

De forma predeterminada, muestra todas las aplicaciones listadas en el archivo de configuración settings en la variable INSTALLED_APPS que se han registrado con la aplicación admin, en orden alfabético. Tal vez lo que quieres es hacer pequeños cambios en el diseño. Después de todo, el índice es probablemente la página más importante de la interfaz administrativa y debería ser fácil de utilizar.

La plantilla que vamos a modificar es admin/index.html. (Hacemos lo mismo que hicimos con admin/base_site.html en la sección anterior – copiarla del lugar por default hasta nuestro directorio de plantillas.). Al editar el fichero, puedes ver una variable llamada app_list que contiene todas las aplicaciones que se han instalado en Django. En lugar de usar todo esto, puedes incrustar y enlazar objetos específicos o

cualquier cosa que consideres mejor. Una vez más no te preocupes si no puedes entender el lenguaje de plantillas –lo cubriremos más detalladamente en el Tutorial 3.

¿Qué sigue?

Cuando te sientas cómodo con el sitio de administración, puedes seguir *con la tercera parte de este tutorial,* para comenzar a trabajar con las vistas públicas de nuestra aplicación de Encuestas.

CAPÍTULO 05

Escribe tu primera aplicación Django, parte 03

Continuamos con la tercera parte de este tutorial. Seguimos con la aplicación de encuestas Web que hemos creado, nos centraremos ahora en crear la interfaz pública usando –"vistas".

Filosofía

Una vista es un "tipo" de página Web en una aplicación Django que generalmente sirve una función específica y tiene un plantilla específico. Por ejemplo, para una aplicación Web tipo Blog, necesitaríamos tener las siguientes vistas:

- Una página de inicio del blog – que muestra las últimas entradas.
- Una Pagina para "detalles" de cada entrada – con un enlace a cada entrada.
- Una página basada en archivos por año – en la que se muestran todas las entradas publicadas en un respectivo año.
- Una página basada en archivos por mes – en la que se muestran todas las entradas publicadas en un respectivo mes.
- Una página basada en archivos por día – en la que se muestran todas las entradas publicadas en un día determinado.
- Comentarios – Maneja los comentarios de una entrada dada.

Para nuestra aplicación de Encuestas, necesitaremos crear las siguientes cuatro vistas:

- Una Página de "índice" – Que mostrara las últimas preguntas publicadas.
- Una página de "detalles" – Que mostrara el texto de una pregunta, con un formulario para votar, pero no mostrara los resultados.
- La página de "resultados" – Que mostrara los resultados para una pregunta en particular.
- Una página para "votar" – Que manejara los votos para una opción determinada en una pregunta en especifico.

En Django, las páginas Web y otros contenidos son manejados por las vistas. Cada vista es representada por una simple función python (o un método, en este caso una

vista basada en clases-base.) Django elige una vista examinando la petición URL (para ser precisos, la parte de la URL después del nombre del dominio)

Hasta ahora, tal vez hayas encontrado en la Web bellezas tales como "ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D 9B65B1B". En vez de esto, estarás contento de saber que Django te permite construir URL elegantes, usando patrones URL.

Un patrón URL es simplemente la forma general de una dirección URL - por ejemplo: /archivos/<año>/<mes>/.

Para llamar la URL a una vista, Django usa lo que conocemos como una 'URLconfs'. Una URLconfs mapea o enlaza un patrón URL (descrito como una expresión regular) a una vista.

Este tutorial provee una instrucción básica acerca del uso de URLconfs, si necesitas mas referencias puedes consultar el modulo django.core.urlresolvers para obtener más información.

Escribe tu primera vista

Vamos a escribir tu primera vista. Abre el archivo encuestas/views.py y pon el siguiente código python en él:

encuestas/views.py

```
from django.http import HttpResponse

def indice(request):
    return HttpResponse("Hola mundo. Este es el índice
        de las Encuestas")
```

Esta es la vista más sencilla posible. Para llamar a esta vista, necesitas mapear la vista a una URL - y para ello es necesaria una URLconf.

Para crear una URLconf en el directorio de tu aplicación encuestas, **crea** un archivo llamado `urls.py`. El directorio de tu aplicación debe parecerse al siguiente:

```
encuestas/
    __init__.py
    admin.py
    migrations/
             __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

En el archivo encuestas/urls.py incluye el siguiente código:

encuestas/urls.py

```
from django.conf.urls import url

from encuestas import views

urlpatterns = [
    url(r'^$', views.indice, name='indice'),
]
```

El siguiente paso es conectar la URLconf a la raíz del el modulo “encuestas.urls”`. En misitio/urls.py inserta la función include(), de la siguiente forma:

misitio/urls.py

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

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

No es lo mismo que vez: Si estás viendo admin.autodiscover() antes de la definición de urlpatterns, lo más seguro es que estés usando una versión distinta a la de este tutorial, puedes cambiar tu versión de Django o usar un tutorial anterior.

Ahora ya tenemos una vista de índice dentro de la URLconf. Dirige tu navegador a la página http://localhost:8000/encuestas/ y podrás ver el texto que dice: *"Hola mundo. Este es el índice de las Encuestas"* el cual definimos en la vista índice.

Se le pueden pasar cuatro argumentos a la función url(), dos requeridos, un **regex** y una ***vista*** y dos opcionales: **kwargs** y **name.** En este punto, es importante repasar estos cuatro argumentos.

url() argumento: regex

El término “regex” es comúnmente usado como una abreviatura para nombrar una **expresión regular**, que es una forma de describir patrones que se verifiquen en una cadena, en este caso patrones de una url. Cuando un usuario solicita una página creada por Django, el sistema busca una cadena en python, Django busca un módulo que contiene una variable llamada urlpatterns que es una secuencia de tuplas con el formato siguiente: (Expresión regular, Llamada a una función Python [, Diccionario opcional])

Observa que estas expresiones regulares no buscan parámetros GET y POST o el nombre del dominio. Por ejemplo, en una petición a http://www.example.com/encuestas/, el URLconf buscará encuestas/. En una petición a http://www.example.com/encuestas/?page=3, la URLconf buscará encuestas/.

Si necesitas más ayuda con las expresiones regulares, puedes consultar la Wikipedia (http://en.wikipedia.org/wiki/Regular_expression) y en la documentación del modulo re de Python También, está el fantástico libro de O'Reilly "Mastering Regular Expressions" de Jeffrey Friedl.

Finalmente, una nota sobre el funcionamiento de las expresiones regulares; estas se compilan la primera vez que son cargadas. Esto hace que sean muy rápidas (mientras que no sean demasiado complejas según lo mencionamos arriba).

url() argumento: view

Cuando Django encuentra una expresión regular que coincide, llama a la función vista especifica, con un objeto HttpRequest como primer argumento y los valores que se hayan "capturado" a partir de la expresión regular como otros argumentos. Si el regex usa capturas simples, estos valores son pasados como argumentos posicionales; si se usan capturas con nombre, los valores se pasan como argumentos clave. Veremos un ejemplo más adelante.

url() argumento: Kwargs

Se pueden pasar argumentos clave arbitrarios en un diccionario a la vista de destino. Aunque no vamos a usar esta característica en este tutorial, esta es muy útil.

url() argumento:name

Darles un nombre a las URL nos permite referirnos a ellas de forma inequívoca y sin ambigüedades desde distintas partes de Django, especialmente en las plantillas. Esta poderosa característica nos permite hacer cambios globales en los patrones url del proyecto, sin tocar más que un solo archivo.

Escribiendo vistas

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

encuestas/views.py

```
def detalles(request, pregunta_id):
    return HttpResponse("Estas buscando la pregunta %s."
        % pregunta_id)

def resultados(request, pregunta_id):
    response = "Estas buscando los resultados de la
        pregunta %s."
    return HttpResponse(response % pregunta_id)

def votar(request, pregunta_id):
    return HttpResponse("Haz votado en la pregunta %s."
        % pregunta_id)
```

Ahora conectemos estas nuevas vistas a el módulo encuestas.urls agregando la siguientes llamada a el método url():

encuestas/urls.py

```
from django.conf.urls import url

from encuestas import views

urlpatterns = [
    # Por ejemplo: /encuestas/
    url(r'^$', views.indice, name='indice'),
    # Por ejemplo: /encuestas/5/
    url(r'^(?P<pregunta_id>[0-9]+)/$', views.detalles,
        name='detalles'),
    # Por ejemplo:/ encuestas /5/resultados/
    url(r'^(?P<pregunta_id>[0-9]+)/resultados/$',
        views.resultados, name='resultados'),
    # Por ejemplo:/encuestas/5/votar/
    url(r'^(?P<pregunta_id>[0-9]+)/votar/$', views.votar,
        name='votar'),
]
```

Ahora abre tu navegador y visita la dirección "/encuestas/34/". Se ejecutara la función `detalles()` y mostrará el ID que le hayas pasado a la URL. Prueba ahora con "/encuestas/34/resultados/" y "/encuestas/34/votar/" – esto debería de mostrar la página de resultados y la de votos respectivamente.

Cuando alguien solicita una página de tu sitio Web – digamos, "/encuestas/34/", Django carga el módulo Python misitio.urls, que apunte al archivo de configuración settings.py en la variable ROOT_URLCONF. Este encuentra la variable llamada urlpatterns y recorre las expresiones regulares en orden. Las llamada a el método include() simplemente hace referencia a otra URLconf. Nota que las expresiones regulares para la función include() no tienen un $ (carácter que indica el fin de una cadena 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 la cadena restante a la URLconf para que sea procesada.

La idea detrás de include() es crear fácilmente URLs reusables. Como encuestas tiene su propia URLconf (encuestas/urls.py), las URLs de la aplicación se pueden poner bajo "/encuestas/" o bajo "/preguntas/", o bajo "/contenido/encuestas/", o cualquier otro camino que desees y la aplicación seguirá funcionando sin problemas.

Esto es lo que pasa cuando un usuario va a "/encuestas/34/" en el sistema:

- Django encontrará coincidencia con '^encuestas/'
- Entonces, Django va a recortar el texto que coincide con ("encuestas/") y enviara el texto restante – "34/" a la URLconf 'encuestas.urls' para seguir el proceso, donde coincidirá con r'^(?P<pregunta_id>[0-9]+)/$' resultando en una llamada a la vista detalles() de esta forma:

  ```
  detalles(request=<HttpRequest object>, pregunta_id='34')
  ```

  La parte pregunta_id='34' viene de (?P<pregunta_id>[0-9]+). Usamos paréntesis alrededor de un patrón para "capturar" el texto que coincida con el patrón y ese valor se pasas como argumento a la función vista;

?P<pregunta_id> define el nombre que se usará para identificar la coincidencia del patrón; y [0-9]+ es una expresión regular para buscar una secuencia de dígitos (por ejemplo, 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 esto:

url(r'^encuestas/ultimas\.html$', views.indice),

Pero, hacer esto es absurdo.

Escribiendo vistas más útiles

Cada vista es responsable de hacer una de dos cosas: Retornar un objeto HttpResponse que contiene el contenido de cada página pedida, o lanzar una excepción por ejemplo Http404 una página no encontrada. El resto depende de ti.

Tus vistas puede leer registros de la base de datos, o no. Puede utilizar el sistema de plantillas de Django – o un sistema de plantillas de terceros – o no. Puedes generar un fichero pdf, crear salidas en XML, crear archivos ZIP al vuelo o cualquier cosa, usando las bibliotecas de Python el límite es la imaginación.

Todo lo Django necesita es un objeto HttpResponse o una excepción.

De forma conveniente, Django te permite utilizar una API, para tener acceso a la base de datos, tal como lo vimos en el *Tutorial 1*. Veamos ahora como mostrar las últimas 5 preguntas publicadas en nuestro sistema, separadas por comas, de acuerdo a la fecha de publicación:

encuestas/views.py

```
from encuestas.models import Pregunta

def indice(request):
    ultimas_preguntas = Pregunta.objects.order_by('-fecha')[:5]
    salida = ', '.join([p.texto_pregunta for p in
       ultimas_preguntas])
    return HttpResponse(salida)

# Dejamos el resto de vistas sin cambios (detalles, resultados…)
```

Tenemos un problema aquí, hemos mezclado la plantilla con la vista, si queremos cambiar la apariencia de la paginas necesitamos editar código Python. Utilicemos el sistema de plantillas para separar el diseño del código python.

Primero, crea un directorio llamado “templates” para las plantillas, en el directorio encuestas, Django buscara las plantillas allí.

La variable TEMPLATE_LOADERS en Django, contiene una lista de llamables que saben cómo importar las plantillas de varios lugares. Uno de estos lugares de forma predeterminada es
django.template.loaders.app_directories.Loader que busca un subdirectorio "templates" en cada aplicación instalada en INSTALLED_APPS – esta es la forma en que Django sabe donde debe buscar las plantillas, siempre y cuando no hayas modificado la variable TEMPLATE_DIRS, como lo hicimos en el *Tutorial 2*.

Organizando las plantillas

Ahora podrías colocar todas tus plantillas juntas, en un gran directorio de plantillas y funcionaría perfectamente bien. Sin embargo, estas plantilla pertenecen a la aplicación de `Encuestas`, por lo que a diferencia de las plantillas de administración que creamos en el tutorial anterior, vamos a poner estas en el directorio de plantillas de la aplicación (encuestas/templates) en lugar del proyecto. Vamos a examinar más a detalle *por qué* hacemos esto en el *tutorial sobre aplicaciones reusables.*

Nota: Colocar juntas todas tus plantillas, en el directorio `encuestas/templates` (En lugar de crear otro subdirectorio encuestas), es en realidad una mala idea. Ya que Django elegirá la primera plantilla que encuentre cuyo nombre coincida, si tuviéramos *diferentes* aplicaciones con nombres de plantillas iguales, Django sería incapaz de distinguir entre ellas, de vemos asegurarnos de que Django encuentre las plantillas correctas y la forma más sencilla de hacer esto, es usando una separación de nombres entre las distintas aplicaciones. Esto es, poniendo las plantillas dentro de otro subdirectorio, con el nombre de la aplicación en sí.

Dentro del directorio templates que acabas de crear, crea otro directorio llamado encuestas y dentro de este, crea un archivo llamado index.html. En otras palabras, la plantilla debe de encontrarse en: encuestas/templates/encuestas/index.html. Debido a la forma en que funciona el buscador de plantillas de app_directories, como se describió anteriormente, se puede hacer referencia a esta plantilla en Django sencillamente como encuestas/index.html.

Usa el siguiente código en la plantilla:

encuestas/templates/encuestas/index.html

```
{% if ultimas_preguntas %}
    <ul>
    {% for pregunta in ultimas_preguntas %}
        <li><a href="/encuestas/{{ pregunta.id }}/">
       {{ pregunta.texto_pregunta }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No hay encuestas publicadas.</p>
{% endif %}
```

Ahora actualicemos la vista de índice en encuestas/views.py para usar la plantilla que hemos creado:

encuestas/views.py

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

from encuestas.models import Pregunta

def indice(request):
    ultimas_preguntas = Pregunta.objects.order_by('-fecha')[:5]
    template = loader.get_template('encuestas/index.html')
    context = RequestContext(request, {
        'ultimas_preguntas': ultimas_preguntas,
    })
    return HttpResponse(template.render(context))
```

Este código carga la plantilla llamada encuestas/index.html y pasa el contexto. El contexto es un diccionario que asocia nombre de variables de plantillas con objetos Python.

Recarga la página una vez más y dirige tu navegador a '/encuestas/'. Ahora puedes ver una lista de viñetas que contiene la pregunta: "¿Qué pasa?" que creamos en el tutorial 1, con un enlace que apunta a la página de detalles de la pregunta.

El atajo: render_to_response ()

Porque es muy común cargar una plantilla, rellenar un contexto y devolver una respuesta Web usando HttpResponse. Django proporciona un atajo. Esta es la vista indice() reescrita totalmente:

encuestas/views.py

```
from django.http import HttpResponse
from django.shortcuts import render
from encuestas.models import Pregunta

def indice(request):
    ultimas_preguntas = Pregunta.objects.all().order_by
       ('-fecha')[:5]
    context = {'ultimas_preguntas': ultimas_preguntas }
    return render(request, 'encuestas/index.html', context)
```

Observa que una vez que hemos usado el atajo `render()` en la vista, no necesitamos importar mas `loader`, `RequestContext` y `HttpResponse` (Seguiremos usando el método `HttpResponse` todavía en las vistas de detalles, resultados y votos).

La función render() toma un objeto request como el primer argumento, una plantilla como segundo argumento y un diccionario como el tercer argumento opcional y devuelve un objeto `HttpResponse` con la plantilla renderizada con el contexto dado.

Lanzando un error 404

Ahora, crea la vista para los detalles de cada pregunta – la página muestra una pregunta para una encuesta dada. Aquí está la vista para los detalles():

encuestas/views.py
```
from django.http import Http404
from django.shortcuts import render

from encuestas.models import Pregunta
# ...

def detalles(request, pregunta_id):
    try:
        pregunta = Pregunta.objects.get(pk= pregunta_id)
    except Pregunta.DoesNotExist:
        raise Http404
    return render(request, 'encuestas/detalles.html',
        {'pregunta': pregunta})
```

El nuevo concepto aquí es que la vista levanta una excepción `Http404` (página no encontrada) si no existe una pregunta con la ID pedido.

Más adelante discutiremos lo que podremos poner en nuestra plantilla encuestas/detalles.html por el momento bastara con usar esto:

encuestas/templates/polls/detalles.html
```
{{ pregunta }}
```

Esto trabajara bien por ahora.

El atajo: get_object_or_404 ()

Es muy común usar el método `get()` y lanzar una excepción `Http404` si el objeto no existe. Django proporciona un atajo muy útil. Este es el la vista detalles(), actualizada usando el atajo `get_object_or_404 ()`:

encuestas/views.py
```
from django.shortcuts import get_object_or_404, render

from encuestas.models import Pregunta
# ...

def detalles(request, pregunta_id):
    pregunta = get_object_or_404(Pregunta, pk= pregunta_id)
    return render(request, 'encuestas/detalles.html',
        {'pregunta': pregunta})
```

La función `get_object_or_404` toma un modelo de Django como primer argumento y un número arbitrario de argumentos clave, que pasa a la función `get()` del mánager del modelo. Lanza una excepción `Http404` si no existe el objeto.

Filosofía: ¿Porqué utilizar una función auxiliar get_object_or_404 () en vez de atrapar automáticamente las excepciones ObjectDoesNotExist en un nivel superior o teniendo la API de modelos que lanzan un error Http404 usamos ObjectDoesNotExist?

Porque eso acoplaría la capa del modelo con la vista. Una de las primeras metas de los diseñadores de Django es mantener el acoplamiento débil. Algunos acoplamientos controlados se introducen en el modulo django.shortcuts.

El modulo django.shortcuts contiene además un método `get_list_or_404` que trabaja como la función get_object_or_404 – excepto que usa filter() en lugar de get() y levanta una excepción Http404 si la lista está vacía.

Usar el sistema de Plantillas

De regreso a la vista detalles() de nuestra aplicación de Encuestas. Démosle el contexto a la variable `pregunta`, esta es la plantilla "encuestas/detalles.html" rescrita:

encuestas/templates/encuestas/detalles.html

```
<h1>{{ pregunta.texto_pregunta }}</h1>
<ul>
{% for opcion in pregunta.opcion_set.all %}
    <li>{{ opcion.texto_opcion }}</li>
{% endfor %}
</ul>
```

El sistema de plantillas utiliza la sintaxis de puntos para acceder a los atributos de una variable por ejemplo {{pregunta.texto_pregunta}}, Django primero hace una consulta a los diccionarios del objeto pregunta. Fallando silenciosamente si no encuentra los atributos de búsqueda. --que trabajan, en este caso. Si el atributo de la búsqueda falla intenta una búsqueda del índice de lista.

El método llamado pasa por el bucle {% for %}, pregunta.opcion_set.all es interpretado como código Python pregunta.opcion_set.all (); el cual retorna un iterable de los objetos opción que pueden ser llamados por la etiqueta {% for %}

Puedes consultar la guía de plantillas para aprender más sobre el tema.

Removiendo URL incrustadas en plantillas

Recuerdas cuando escribimos el enlace a una pregunta en la plantilla `encuestas/index.html`, el enlace fue parcialmente incrustado así:

```
<li><a href="/encuestas/{{ preguntas.id }}/">
    {{ pregunta.texto_pregunta}}</a></li>
```

El problema con este acercamiento de incrustar el código en las plantillas, es que acoplamos las plantillas y las URLs. Este enfoque se convierte en un desafío al cambiar las URL en los proyectos con una gran cantidad de plantillas. Sin embargo, ya que hemos definido el argumento **`name`** en cada url() de el modulo `encuestas.urls`, podemos eliminar las pendencias de las rutas URL específicas, con el argumento **`name`** que hemos definido en la configuración de la URL, usando la etiqueta de plantilla `{% url %}`.

```
<li><a href="{% url 'detalles' pregunta.id %}">
    {{ pregunta.texto_pregunta}}</a></li>
```

La forma en que esto funciona es buscando la definición de la URL que se especifico en el modulo encuestas.url, podemos ver exactamente donde se definió el **name** (nombre) de la URL en la URLconf para 'detalles' a continuación:

```
...
#El valor 'name' es llamado por la etiqueta {% url %}
url(r'^(?P<pregunta_id>\d+)/$',views.detalles, name='detalles'),
...
```

Si quieres cambiar la URL de encuestas de la vista detalles, por algo un poco diferente como encuestas/especificas/12/ en lugar de hacerlo en la plantilla (o plantillas) solo debes cambiar el archivo en la url, como por ejemplo: encuestas/urls.py:

```
...
# Agrega la palabra 'especifica'

    name='detalles'),
...
```

Espacios de nombres en URL

El proyecto de este tutorial contiene una sola aplicación; `Encuestas`. En proyectos reales de Django, podrías tener cinco, diez, veinte o más aplicaciones. ¿Cómo distingue Django los nombres de las URL entre ellos? Por ejemplo, la aplicación encuestas tiene una vista detalles, por lo que podrías tener otra aplicación, por ejemplo un blog con el mismo nombre. ¿Cómo hace Django para saber que aplicación va usar en la vista cuando se utiliza la etiqueta de plantilla {% url%}.

La respuesta consiste en agregar **namespace** (espacios de nombres) a la raíz de la URLconf. Para seguir adelante agreguémosle **namespace** a el archivo misitio/urls.py (el archivo del proyecto urls.py, no el de la aplicación)

misitio/urls.py

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

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

Ahora cambia la plantilla de encuestas/index.html, que escribimos así:

encuestas/templates/encuestas/index.html

```
        <li><a href="{% url 'detalles' pregunta.id %}">{{
               pregunta.texto_pregunta }}</a></li>
```

Para que apunte a la vista detalles con el espacio de nombres así:

encuestas/templates/encuestas/index.html

```
    <li><a href="{% url 'encuestas:detalles' pregunta.id %}">
              {{pregunta.texto_pregunta }}</a></li>
```

¿Qué sigue?

Cuando te sientas cómodo escribiendo vistas, puedes leer la cuarta parte de este tutorial para aprender sobre procesamiento de formularios y sobre las vistas genéricas.

CAPÍTULO 06

Escribe tu primera aplicación Django, parte 04

Continuamos con la cuarta parte de este tutorial. Seguimos construyendo una aplicación de encuestas Web y en esta parte nos centraremos en la forma de procesar formularios sencillos y en escribir menos código.

Escribe un formulario sencillo

Vamos a actualizar la plantilla detalles.html que creamos en el tutorial anterior ("encuestas/detalles.html") para incluir un formulario usando un elemento <form> de HTML.

encuestas/templates/encuestas/detalles.html

```
<h1>{{ pregunta.texto_pregunta }}</h1>

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

<form action="{% url 'encuestas:votar' pregunta.id %}"
   method="post">
{% csrf_token %}
{% for opcion in pregunta.opcion_set.all %}
   <input type="radio" name="opcion"
   id="opcion{{forloop.counter }}" value="{{ opcion.id }}" />
   <label for="opcion{{ forloop.counter }}">
   {{opcion.texto_opcion }}</label><br />
{% endfor %}
<input type="submit" value="Votar" />
</form>
```

Un breve resumen:

- La plantilla anterior muestra un botón para cada una de las opciones de la encuesta. El valor de cada radio botón está asociado a él ID de cada opción de una pregunta. El nombre de cada botón es "opcion". Esto significa que cuando alguien selecciona uno de los radio botones, nos envía los datos a través de un formulario, usando **POST** con los datos opción = #donde# es el ID de la opción seleccionada. Éste es el concepto básico de un formulario HTML.

- Hemos ajustado las acciones del formulario, estableciendo action a: {% url 'encuestas:votar' encuestas.id %} y establecido el método method = "post" . Usamos el method = "post" ` (en forma opuesta al method = "get")

esto es muy importante, porque el acto de subir un formulario a un servidor puede alterar los datos del lado del servidor. Siempre que creamos un formulario para subir datos, debemos asegurarnos de usar el method="post". Esta práctica no se especifica de Django; es una buena práctica de desarrollo Web.

- forloop.counter indica cuántas veces la etiqueta for ha sido usada en un bucle.

- Puesto que estamos creando un formulario con peticiones **POST** (que puede tener el efecto de modificar datos), necesitamos preocuparnos de las Falsificación de peticiones inter-sitio o "*Cross Site Request Forgeries*". Gracia a Django, no tenemos que preocuparnos demasiado, porque viene con un sistema muy fácil de utilizar para protegernos contra ellas. En fin, todos los formularios **POST** que apunten a una URL interna deben utilizar la etiqueta {% csrf_token %}

Ahora, veamos como las vistas de Django manejan los datos enviados y hacen algo útil con ellos. Recuerdas que en el *Tutorial 3*, creamos una URLconf para la aplicación encuestas que incluía la línea siguiente:

encuestas/urls.py

```
url(r'^(?P<pregunta_id>[0-9]+)/votar/$', views.votar,
    name='votar'),
```

También habíamos creado una función "temporal" que implementaba la función votar(). Ahora es tiempo de crear una versión real. Agrega lo siguiente al archivo encuestas/views.py:

encuestas/views.py

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse

from encuestas.models import Pregunta, Opcion
# ...

def votar(request, pregunta_id):
    p = get_object_or_404(Pregunta, pk=pregunta_id)
    try:
        seleccionar_opcion=p.opcion_set.get(
           pk=request.POST['opcion']) except (
               KeyError, Opcion.DoesNotExist):
        # Vuelve a mostrar el formulario con un mensaje de error
        return render(request, 'encuestas/detalles.html', {
            'pregunta': p,
            'error_message': "No has seleccionado una opción.",
        })
    else:
        seleccionar_opcion.votos += 1
        seleccionar_opcion.save()
        # Siempre devuelve una respuesta HttpResponseRedirect
        # después de enviar los datos con éxito usando POST.
```

```
        # Esto evita que los datos sean posteados dos veces. Si
        # el usuario usan el botón de  retroceso
        return HttpResponseRedirect(reverse(
            'encuestas:resultados', args=(p.id,)))
```

Este código incluye algunas cosas que no hemos cubierto en este tutorial:

- **request.POST** es un objeto tipo-diccionario que te permite acceder a datos a través del tipo de clave de su nombre. En este caso, request.**POST** ["opcion"] retorna la ID de la opción seleccionada, como una cadena. Los valores request.**POST** son siempre cadenas.
  Observa que Django siempre proporciona un request.GET para tener acceso a los datos de la misma forma –pero estamos utilizando explícitamente request.**POST**, para asegurarnos que los datos solamente puedan ser alterados mediante una llamada a **POST**.

- **request.POST** [“opcion”] levanta una excepción “KeyError” si no se le provee datos mediante **POST**. El código de arriba volverá a mostrar el formulario de preguntas con un mensaje de error si no le proporcionamos una opción.

- Después de incrementarse el contador de opciones, el código retorna una HttpResponseRedirect antes que una normal respuesta HttpResponse. HttpResponseRedirect toma un solo argumento: la URL a la cual se volverá a redireccionar al usuario (véase el siguiente punto para ver cómo construimos la URL en este caso).

  Como un comentario extra fuera de Python, en este punto debes siempre retornar un HttpResponseRedirect después de haber subido exitosamente datos mediante **POST**. Este consejo no es específico de Django; es simplemente una buena práctica de desarrollo Web en general.

- Estamos usando la función reverse() de la clase HttpResponseRedirect como el constructor en este ejemplo. Esta función nos ayuda a evitar mezclar la URL en la función vista. Simplemente se le pasa el nombre de la vista a la que queremos pasar el control y la parte de la variable del patrón al que apunta la URL a esa vista. En este caso usamos la URLconf que diseñamos en el tutorial 3, esta función llama reverse() y retorna una cadena como esta:

  "/encuestas/3/resultados/"

  ...donde 3 es el valor de p.id. Este redirecciona la URL y llama a la vista "resultados" la vista muestra la pagina final.

Según lo mencionado en el tutorial 3, la petición request es una clase de HttpRequest.

Después de que alguien vota en una pregunta, la vista votar() re-direcciona a los usuarios a una página de resultados para la pregunta. Escribe la vista para los resultados:

encuestas/views.py
```
from django.shortcuts import get_object_or_404, render

def resultados(request, pregunta_id):
    pregunta = get_object_or_404(Pregunta, pk=pregunta_id)
    return render(request, 'encuestas/resultados.html',
         {'pregunta': pregunta})
```

Ahora crea la plantilla encuestas/resultados.html:

encuestas/templates/encuestas/resultados.html
```
<h1>{{ pregunta.texto_pregunta }}</h1>

<ul>
{% for opcion in pregunta.opcion_set.all %}
    <li>{{ opcion.texto_opcion }} -- {{ opcion.votos }}
    voto{{ opcion.votos|pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'encuestas:detalles' pregunta.id %}">
¿Quieres votar otra vez?</a>
```

Dirígete a /encuestas/1/ en tu navegador y vota en una encuesta. Deberías ver una página de resultados que se actualiza cada vez que votas. Si subes un formulario sin seleccionar una opción, te mostrara un mensaje de error.

Usa las vistas genéricas: Menos código es mejor

Las vistas detalles() del *Tutorial 3* y la vista resultados() son extremadamente simples – y, según lo mencionamos anteriormente, redundantes. La vista para el índice (también del tutorial 3), que visualiza la lista de encuestas, es muy similar.

Estas vistas representan un caso común en el desarrollo web: obtener los datos de una base de datos según un parámetro dado en una URL, cargar una plantilla y retornar una plantilla renderizada. Porque esto es tan común, Django proporciona un sencillo atajo, llamar al sistema de “vistas genéricas”.

Las vistas genéricas abstraen los patrones comunes a tal punto que incluso no necesitas escribir código python para escribir una aplicación.

Bien, vamos a modificar nuestra aplicación de encuestas para utilizar el sistema de vistas genéricas de Django, así que podemos suprimir un montón de nuestro propio código. Solo tendremos que tomar algunas medidas para realizar la conversión:

1. Convertir las URLconf.
2. Borrar las vistas innecesarias.
3. Introducir las nuevas vistas, basadas en vistas genéricas.

Veamos los detalles.

¿Por qué mezclar código? Generalmente, al escribir una aplicación, es necesario evaluar si las vistas genéricas son una buena idea para resolver nuestro problema y así poder utilizarlas desde el principio, sin tener que refactorizar el código a medio camino. Pero esta guía se ha centrado intencionalmente en escribir vistas "*de la manera difícil*" hasta ahora, nos hemos enfocado en los conceptos base para entender cómo funciona Django.

Debes saber matemáticas antes de aprender a usar una calculadora.

Edita la ULRconf

Primero, edita el archivo encuestas/urls.py de la URLconf de esta manera:

encuestas/urls.py

```
from django.conf.urls import url

from encuestas import views

urlpatterns = [
    url(r'^$', views.VistaIndice.as_view(), name='indice'),
    url(r'^(?P<pk>[0-9]+)/$', views.VistaDetalles.as_view(),
        name='detalles'),
    url(r'^(?P<pk>[0-9]+)/resultados/$',
        views.VistaResultados.as_view(), name='resultados'),
    url(r'^(?P<pregunta_id>[0-9]+)/votar/$', views.votar,
        name='votar'),
]
```

Observa que el nombre de patrón de coincidencias en el regex del segundo y tercer patron ha cambiado de <pregunta_id> a < pk> .

Edita las vistas

A continuación, vamos a remover las vistas que usamos anteriormente `índice`, `detalles` y `resultados` para utilizar las vistas genéricas de Django en su lugar. Para ello, abre el archivo `encuestas/views.py` y cámbialo así:

encuestas/views.py

```
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 encuestas.models import Pregunta, Opcion

class VistaIndice(generic.ListView):
    template_name = 'encuestas/index.html'
    context_object_name = 'ultimas_preguntas'

    def get_queryset(self):
```

```
        """Devuelve las ultimas 5 preguntas publicadas."""
        return Pregunta.objects.order_by('-fecha')[:5]

class VistaDetalles(generic.DetailView):
    model = Pregunta
    template_name = 'encuestas/detalles.html'

class VistaResultados(generic.DetailView):
    model = Pregunta
    template_name = 'encuestas/resultados.html'


def votar(request, question_id):
… # Esta función no cambia
```

En el ejemplo anterior, estamos usando dos vistas genéricas: ListView y DetailView. Respectivamente, estas dos vistas abstraen el concepto de mostrar una lista de objetos: ListView y mostrar los detalles de un determinado objeto: DetailView.

- Cada vista genérica necesita saber sobre qué modelo actuará. Esto se proporciona usando el atributo model.

- La vista genérica DetailView espera capturar el valor primario de la URL que se llamará "pk", así que hemos cambiado el pregunta_id por pk para usar la vista genérica.

De forma predeterminada la vista genérica de la clase DetailView utiliza una plantilla que toma su nombre de <nombre-aplicación>/<nombre-modelo>_details.html. En nuestro caso, utilizará la plantilla "encuestas/detalles.html". El argumento template_name se utiliza para decirle a Django que utilice una plantilla en específico, en vez del nombre generado automáticamente. También especificamos un atributo template_name para la vista resultados –con esto nos aseguramos que las vista resultados y la vista detalles tengan una apariencia distinta cuando sean renderizadas, aunque ambas usen la misma vista DetailView detrás de escena.

Similarmente la vista genérica de la clase ListView usan una plantilla por defecto llamada <nombre-aplicación>/<nombre-modelo>_list.html; hemos usado el atributo template_name para indicarle a la vista ListView que use la plantilla "encuestas/index.html".

Anteriormente en este tutorial, le habíamos proporcionado un contexto en las vistas a nuestras plantillas, la variable de contexto: pregunta y ultimas_preguntas. Para DetailView la variable pregunta se genera automáticamente –porque estamos usando el modelo Django (Pregunta), Django puede determinar un nombre apropiado para la variable del contexto. Sin embargo, para ListView, la variable automáticamente generada del contexto era pregunta_list. Para sobre-escribir esto proporcionamos el atributo context_object_name especificando que queremos utilizar ultimas_preguntas en lugar de preguntas_list. Como alternativa, podríamos cambiar las plantillas para que concuerden con las nuevas variables por defecto –pero es más fácil decirle a Django que utilice solo la variable que queremos usar.

Reinicia el servidor y podrás utilizar la nueva aplicación de encuestas basada en vistas genéricas.

Para más detalles acerca del uso de vistas genéricas, consulta el capitulo “Las vistas basadas en clases genéricas”

¿Qué sigue?

Cuando te sientas cómodo usando las vistas genéricas y los formularios, puedes leer la quinta parte de este tutorial, para aprender cómo poner a prueba nuestra aplicación de encuestas, usando “test’s”.

CAPÍTULO 07

Escribe tu primera aplicación Django, parte 05

Continuamos con la quinta parte de este tutorial. Hemos construido una aplicación de encuestas Web y ahora vamos a agregar algunas pruebas automáticas para la misma.

Introducción a pruebas automatizadas

¿Qué son las pruebas automatizadas?

Las pruebas o "tests" son sencillas rutinas que verifican el funcionamiento de tu código.

Las pruebas operan en diferentes niveles. Algunos de las pruebas pueden aplicarse a un pequeño detalle (*a un método en particular de un modelo que devuelve el valor esperado*), mientras que otros examinan el funcionamiento en general del software - (donde una secuencia de entradas de algún usuario produce el resultado deseado) No son diferentes del tipo de pruebas que hicimos en el *Tutorial 1*, usando el shell interactivo para examinar el comportamiento de un método o al ejecutar la aplicación e ingresar datos para comprobar cómo se comportan.

Lo que hace diferentes a los **`tests automatizados`** es que el trabajo de hacer las pruebas lo hace el sistema por ti. Tu solo creas un conjunto de pruebas una vez y luego a medida que se hacen cambios en la aplicación, se puede verificar que el código sigue todavía funcionando como estaba pensado originalmente, sin tener que perder tiempo haciendo el testeo manualmente.

¿Por qué es necesario crear tests?

¿Por qué crear pruebas 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 funciona bien; 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 necesitas saber cómo crear tests automatizados. Pero si no es el caso, este es un excelente momento para aprender.

Los tests ahorrarán tu tiempo

Hasta cierto punto, '**comprobar que todo funciona bien'** en una prueba es satisfactorio. En una aplicación más sofisticada, podrías tener una docena de interacciones complejas entre los componentes.

Un cambio en cualquiera de estos componentes podría tener consecuencias inesperadas en el comportamiento de la aplicación. Comprobar que todavía 'parece

que funciona' podría significar que debemos analizar el funcionamiento del código con veinte variaciones diferentes de datos y pruebas solo para estar seguros de que no se haya roto algo -- no es un buen uso de tu tiempo.

Esto es particularmente cierto cuando usas tests automatizados que podrían hacerlo por ti en segundos. Si algo se rompe, los tests te ayudaran a identificar el código que está causando el comportamiento inesperado.

Algunas veces puede parecer una tarea que nos distrae de nuestro trabajo creativo 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 pruebas es mucho más gratificante que gastar horas probando la aplicación manualmente o intentando identificar la causa de un problema recién introducido.

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 nuestro propio código, 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 tu código más atractivo

Podrías crear una pieza brillante de software, pero muchos desarrolladores simplemente se van a rehusar a usarla, 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 buena razón para empezar a escribir tests.

Los tests ayudan a trabajar en equipo

Los puntos anteriores están escritos desde el punto de vista de un único desarrollador manteniendo una aplicación. Sin embargo en aplicaciones complejas que son mantenidas por equipos. Los tests garantizan que otros colegas no rompan tu 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

Existen muchas maneras de aprender a escribir tests.

Algunos programadores siguen una disciplina llamada "test-driven development", (desarrollo dirigido por tests, busca más información sobre el tema en http://en.wikipedia.org/wiki/Test-driven_development); los tests se escriben antes de escribir el código. Puede parecer contrario a la intuición, pero en realidad es similar a lo que la mayoría de la gente a hace a menudo: describen un problema, luego crean el código que lo resuelve. Basado en pruebas, "Test-driven development" simplemente formaliza el problema como un caso de testeo en Python.

Ocurre muchas veces, por ejemplo, que algún principiante en "testing" crea el código y más tarde decide que debe escribir algunos tests. Tal vez hubiera sido mejor escribir los tests antes, sin embargo nunca es tarde para empezar.

Algunas veces es difícil darse cuenta por dónde empezar a escribir tests. Si uno escribió cientos o varias miles de líneas de código Python, elegir qué testear puede no ser tan sencillo. En ese caso, es más provechoso escribir el primer test la próxima vez que hagas un cambio en tu código, ya sea que agregues una nueva funcionalidad o corrijas un pequeño error.

Así es que hagámoslo de inmediato

Escribe tu primer test

Identificamos un error

Afortunadamente, existe un pequeño error en la aplicación `encuestas` que podemos arreglar: el método fue_publicada_recientemente() del modelo `Pregunta`, devuelve True si una Pregunta fue publicada el día anterior (que está bien), pero también si la fecha está en el futuro (que no está bien!).

Podemos verlo en el Admin; crea una nueva pregunta cuya fecha sea en el futuro; te darás cuenta que la lista de cambios nos dice que fue publicada recientemente.

También podemos verlo usando el shell:

```
>>> import datetime
>>> from django.utils import timezone
>>> from encuestas.models import Pregunta
>>> # Crea una instancia de Pregunta con fecha 30 días en el
>>> # futuro
>>>futura_pregunta= Pregunta(fecha=timezone.now() +
       datetime.timedelta(days=30))
>>> # ¿Fue publicada recientemente?
>>> futura_pregunta.fue_publicada_recientemente()
True
```

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

Crea un test que exponga el error

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

El mejor lugar para colocar los tests de una aplicación es el archivo tests.py- el sistema de pruebas automáticamente buscara tests en cualquier archivo cuyo nombre comience con test.

Coloca el siguiente código en el archivo tests.py de la aplicación encuestas:

encuestas/tests.py

```
import datetime

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

from encuestas.models import Pregunta

class MetodoPruebaPreguntas(TestCase):

    def test_fue_publicada_recientemente_pregunta_futuro(self):
```

```
        """
        fue_publicada_recientemente() debe de devolver
        “False” con preguntas cuya fecha este en el futuro.
        """
        tiempo = timezone.now() + datetime.timedelta(days=30)
        pregunta_futura = Pregunta(fecha=tiempo)
        self.assertEqual(
            pregunta_futura.fue_publicada_recientemente(),
                False)
```

Lo que hemos hecho aquí es crear una subclase de `django.test.TestCase` usando un método que crea una instancia de `Pregunta` con el valor de la fecha en el futuro. Luego verificamos la salida de `fue_publicada_recientemente()` - que *debería* ser False.

Ejecutando pruebas

En la terminal, podemos ejecutar el test así:

```
$ python manage.py test encuestas
```

Y deberías ver algo como lo siguiente:

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_fue_publicada_recientemente_pregunta_futuro
(encuestas.tests.MetodoPruebaPreguntas)
----------------------------------------------------------------------
------
Traceback (most recent call last):
  File   "/ruta/a/misitio/encuestas/tests.py",   line   17,   in
test_fue_publicada_recientemente_pregunta_futuro

self.assertEqual(pregunta_futura.fue_publicada_recientemente(),
    False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.005s

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

Lo que sucedió fue esto:

- El comando python manage.py test encuestas busca tests en la aplicación encuestas.
- Si encuentra una subclase de la clase django.test.TestCase:

- Crea una base de datos especial, para usarla en el testeo.
- Busca métodos para los test - aquellos cuyo nombre comienza con test

- En prueba_para_publicada_recientemente_en_el_futuro creó una instancia de Pregunta cuyo valor para el campo fecha es 30 días en el futuro
- ... y usa el método assertEqual() para descubrir si el método test_fue_publicada_recientemente_pregunta_futuro()devuelve True, a pesar de que queremos que devuelva False.

El informe de pruebas nos informa qué la prueba falló, incluso nos muestra la línea en la que se produjo la falla.

Corrigiendo el error

Ya sabemos cuál es el problema, el método fue_publicada_recientemente() debe devolver False, si el campo fecha tiene un valor en el futuro. Corregimos el método en el archivo models.py, para que sólo devuelva True si la fecha está en el pasado:

encuestas/models.py

```
def fue_publicada_recientemente(self):
    hoy = timezone.now()
    return hoy - datetime.timedelta(days=1) <= self.fecha
        <= hoy
```

Y ejecuta nuevamente la prueba:

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

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

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

Muchas otras cosas pueden salir mal con nuestra aplicación en el futuro, pero podemos estar seguros de una cosa, no vamos a reintroducir este error inadvertidamente, porque simplemente al ejecutar el test nos avisaría de inmediato. Podemos considerar esta parte de nuestra aplicación funcionando y cubierta por siempre (bueno no tanto).

Pruebas más exhaustivas

Mientras estamos realizando pruebas, podemos mejorar la cobertura del método fue_publicada_recientemente() de hecho, sería vergonzoso si hubiéramos arreglado un error y al mismo tiempo introducido uno nuevo.

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

encuestas/tests.py

```
def test_fue_publicada_recientemente_pregunta_pasado(self):
    """
    fue_publicada_recientemente() debe de devolver “False” con
    preguntas que tengan más de un día publicadas.
    """
    tiempo = timezone.now() - datetime.timedelta(days=30)
    pregunta_anterior = Pregunta(fecha=tiempo)
    self.assertEqual(
        pregunta_anterior.fue_publicada_recientemente(), False)

def test_fue_publicada_recientemente_pregunta_reciente(self):
    """
    fue_publicada_recientemente debe de devolver “True” con
    preguntas cuya fecha sea reciente (menor a una hora).
    """
    tiempo = timezone.now() - datetime.timedelta(hours=1)
    pregunta_reciente = Pregunta(fecha=tiempo)
    self.assertEqual(
        pregunta_reciente.fue_publicada_recientemente(), True)
```

Y ahora tenemos tres tests que confirman que fue_publicado_recientemente() devuelve valores sensibles a preguntas con fechas en el pasado, recientes y futuro.

De nuevo, la aplicación encuestas 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 estos tests se comportará de la manera esperada.

Testea una vista

La aplicación encuestas no discrimina: va a publicar cualquier encuesta, no importando la fecha, incluyendo aquellas cuyo campo fecha tenga un valor en el futuro. Deberíamos mejorar esto. Tener un campo `fecha` en el futuro debería significar que la pregunta será pública hasta ese momento, pero mientras tanto debe permanecer invisible a los visitantes.

Un test para una vista

Cuando arreglamos el error anterior, escribimos el 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, necesitamos verificar el comportamiento como lo experimentaría un usuario usando su navegador.

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

El cliente para test's en Django

Django provee un cliente para realizar test's llamado `Client`, que simula la interacción del usuario, con el código a nivel de vistas. Podemos usarlo en tests.py o incluso en el shell.

Empezaremos de nuevo con el shell interactivo, pero antes necesitamos hacer un par de cosas que no serán necesarias en `tests.py`. La primera es crear las variables de entorno en el shell:

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

El método `setup_test_environment()` instala una plantilla que renderizara, lo que te permitirá examinar algunos atributos adicionales en las respuestas como `response.context` que de otro modo no estaría disponible. Ten en cuenta que este método *no* configura la base de datos de prueba, por lo que se llevará a cabo contra la base de datos existente y la salida pueden variar ligeramente en función de las encuestas que hayas creado.

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

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

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

```
>>> # Obtiene  una respuesta de '/'
>>> respuesta = client.get('/')
>>> # Deberíamos esperar un error 404 de esta dirección
>>> respuesta.status_code
404
>>> # Deberíamos esperar encontrar algo en '/encuestas/'
>>> # Usamos 'reverse()' en lugar de mezclar la URL.
>>> from django.core.urlresolvers import reverse
>>> respuesta = client.get(reverse('encuestas:indice'))
>>> respuesta.status_code
200
>>> respuesta.content
```

```
'\n\n\n    <p>No hay encuestas publicadas.</p>\n\n'
>>># Aquí deberían mostrarse todas las preguntas disponibles.
   # Nota –Podrías tener resultados inesperados si ``TIME_ZONE``
   # en ``settings.py`` no es correcto. Si necesitas cambiarlo.
   # será necesario que reinicies la sesión del shell.
>>> from encuestas.models import Pregunta
>>> from django.utils import timezone
>>> # Creamos una pregunta y la guardamos
>>> p = Pregunta(texto_pregunta="¿Quien es tu Beatle favorito?",
        fecha=timezone.now())
>>> p.save()
>>> # Comprobamos la respuesta otra vez.
>>> respuesta = client.get('/encuestas/')
>>> respuesta.content
'\n\n\n  <ul>\n  \n <li><a href="/encuestas/1/"> ¿Quien es tu
Beatle favorito?",
a></li>\n    \n    </ul>\n\n'
>>> respuesta.context['ultimas_preguntas']
[<Pregunta: ¿Quien es tu Beatle favorito?>]
```

Mejorando nuestra vista

La lista de encuestas nos muestra preguntas que no están publicadas todavía (por ejemplo: aquellas que tienen fecha en el futuro). Arreglémoslo.

En el *Tutorial 4* reemplazamos las funciones de vistas en views.py por listas basadas en clases

encuestas/views.py

```
class VistaIndice(generic.ListView):
    template_name = 'encuestas/index.html'
    context_object_name = 'ultimas_preguntas'

    def get_queryset(self):
        """Devuelve las ultimas 5 preguntas publicadas."""
        return Pregunta.objects.order_by('-fecha')[:5]
```

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

Necesitamos modificar el método get_queryset y cambiarlo de modo que también compruebe la fecha, comparándola con la función `timezone.now()`. En primer lugar tenemos que añadir primero la importación

encuestas/views.py

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

Y luego modifica el método `get_queryset así`:

encuestas/views.py

```
def get_queryset(self):
    """
    Devuelve las últimas 5 preguntas publicadas.
    (no incluye aquellas que tienen una fecha en el futuro)
    """
    return Pregunta.objects.filter(fecha__lte=timezone.now()
        ).order_by('-fecha')[:5]
```

Pregunta.objects.filter(fecha__lte=timezone.now()) devuelve un queryset que contiene las instancias de Pregunta cuyo campo fecha es menor o igual que - timezone.now.

Testeando nuestra nueva vista

Ahora podemos verificar que la vista se comporta como esperamos, ejecuta el servidor de desarrollo, cargando el sitio en el navegador, crea algunas preguntas con fechas en el pasado y en el futuro y verifica que solamente se muestren aquellas que han sido publicadas en el pasado. No necesitas repetir estos pasos *cada vez que se hace un cambio que podría afectar esto* - así que vamos a crear una prueba, basándonos en la sesión de el shell anterior.

Agrega lo siguiente a encuestas/tests.py:

encuestas/tests.py

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

Y crea una función para crear preguntas (un atajo), así como también una nueva clase para realizar pruebas:

encuestas/tests.py

```
def crear_pregunta(texto_pregunta, dias):
    """
    Crea una pregunta con el “texto_pregunta” publicada con el
    número de días a partir de ahora(negativo para preguntas
    publicadas en el pasado, positivo para preguntas que no han
    sido publicadas todavía)
    """
    tiempo = timezone.now() + datetime.timedelta(days=dias)
    return Pregunta.objects.create(texto_pregunta=texto_pregunta
                                   ,fecha=tiempo)

class PruebaVistaPregunta(TestCase):
    def test_vista_indice_sin_preguntas(self):
        """
        Si no existen preguntas muestra un mensaje.
        """
        respuesta = self.client.get(reverse('encuestas:indice'))
        self.assertEqual(respuesta.status_code, 200)
        self.assertContains(respuesta, "No hay encuestas
             publicadas.")
```

```
        self.assertQuerysetEqual(
             respuesta.context['ultimas_preguntas'], [])

    def test_vista_con_preguntas_en_el_pasado(self):
        """
        Las preguntas con fecha en el pasado se deben de mostrar
        en la página de índice.
        """
        crear_pregunta(texto_pregunta="Pregunta en pasado.",
            dias=-30)
        respuesta = self.client.get(reverse('encuestas:indice'))
        self.assertQuerysetEqual(
            respuesta.context['ultimas_preguntas'],
            ['<Pregunta: Pregunta en pasado.>'])

    def test_vista_con_preguntas_en_el_futuro(self):
        """
        Las preguntas con fecha en el futuro no se deben de
        mostrar en la página de índice.
        """
        crear_pregunta (texto_pregunta="Pregunta en Futuro.",
            dias=30)
        respuesta = self.client.get(reverse('encuestas:indice'))
        self.assertContains(respuesta, "No hay encuestas
             publicadas.")
        self.assertQuerysetEqual(
            respuesta.context['ultimas_preguntas'], [])

    def test_vista_con_preguntas_en_pasado_y_futuro(self):
        """
        Siempre que existan preguntas en el pasado y el futuro,
        mostrara únicamente las preguntas en pasado.
        """
        crear_pregunta(texto_pregunta="Pregunta en pasado.",
            dias=-30)
        crear_pregunta (texto_pregunta="Pregunta en futuro.",
            dias=30)
        respuesta = self.client.get(reverse('encuestas:indice'))
        self.assertQuerysetEqual(
            respuesta.context['ultimas_preguntas'],
            ['<Pregunta: Pregunta en pasado.>']
        )

    def test_vista_indice_con_dos_preguntas_en_pasado(self):
        """
        La página de índice de preguntas puede mostrar múltiples
        preguntas.
        """
        crear_pregunta(texto_pregunta="Pregunta en pasado 1.",
            dias=-30)
        crear_pregunta(texto_pregunta="Pregunta en pasado 2.",
            dias=-5)
```

```
        respuesta = self.client.get(reverse('encuestas:indice'))
        self.assertQuerysetEqual(
            respuesta.context['ultimas_preguntas'],
            ['<Pregunta: Pregunta en pasado 2.>',
            '<Pregunta: Pregunta en pasado 1.>'])
```

Vamos a mirar más detenidamente el código anterior:

- Primero está la función, crear_pregunta (un atajo) para evitar repetir el proceso de crear preguntas.
- vista_indice_sin_preguntas no crea preguntas, pero se encarga de mostrar el mensaje “No hay encuestas disponibles.” y si ultimas_preguntas está vacío. Nota que la clase django.test.TestCase provee algunos métodos de aserción adicionales. En estos ejemplos usamos assertContains() y el método assertQuerysetEqual().
- Con la función prueba_vista_con_preguntas_en_el_pasado, creamos una pregunta y verificamos que aparezca en la pagina del listado del índice.
- Con la función prueba_vista_con_preguntas_en_el_futuro, creamos una encuesta con una fecha en el futuro. La base de datos se resetea para cada método de pruebas, por lo que la primera encuesta ya no existe y entonces nuevamente no deberías tener ninguna entrada en el listado.

Y así sucesivamente. En efecto, estamos usando las pruebas para contar una historia, mediante entradas en administrador y la experiencia del usuario en el sitio y verificando que en cada estado y para cada nuevo cambio en el estado del sistema, se publiquen los resultados esperados.

Testeando la vista DetailView

Lo que tenemos hasta ahora funciona bien; sin embargo, aunque las preguntas en el futuro no aparecen en la página de *índice*, un usuario todavía puede verlas si sabe o adivina la URL correcta. Necesitamos agregar restricciones similares para la vista `DetailView`, para lo cual agregamos lo siguiente:

encuestas/views.py

```
class VistaDetalles(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excluye cualquier pregunta que no haya sido publicada ya
        """
        return Pregunta.objects.filter(
            fecha__lte=timezone.now())
```

Y por supuesto, vamos a agregar algunos tests para verificar que una instancia de Pregunta con fecha en el pasado se muestra, pero de igual forma una pregunta con fecha en el futuro no:

encuestas/tests.py

```
class PruebaIndiceVistaDetalles(TestCase):
    def test_vista_detalles_con_preguntas_en_el_futuro(self):
       """
        La vista detalles de una pregunta con una fecha en el
        fututo debe mostrar una página no encontrara 404.
        """
        pregunta_futura = crear_pregunta(
            texto_pregunta='Pregunta en el futuro.', dias=5)
        respuesta = self.client.get(reverse(
            'encuestas:detalles', args=( pregunta_futura.id,)))
        self.assertEqual(respuesta.status_code, 404)

    def test_vista_detalles_con_preguntas_en_el_pasado(self):
        """
        La vista detalles de una pregunta con una fecha en el
        pasado debe de  mostrar el texto de la pregunta.
        """
        pregunta_pasado = crear_pregunta(texto_pregunta=
           'Pregunta en pasado.', dias=-5)
        respuesta = self.client.get(reverse(
            'encuestas:detalles', args=( pregunta_pasado.id,)))
        self.assertContains(respuesta,
            pregunta_pasado.texto_pregunta, status_code=200)
```

Ideas para más tests

Deberíamos agregar un método similar a `get_queryset` para la vista `DetailView` y crear una nueva prueba para esta vista. Sería parecido a lo que hemos hecho ya; de hecho, habría bastante repetición.

Podríamos mejorar nuestra aplicación de otras maneras, agregando tests en el camino. Por ejemplo, es tonto permitir que las preguntas que no tengan opciones se puedan publicar en nuestro sitio. Entonces, nuestras vistas podrían verificar eso y también podrían excluir esas preguntas. Nuestros tests crearían una pregunta sin opciones y luego verificarían que no se publique, así como también crearían una pregunta con opciones y verificarían que sí se publique.

Quizás también los administradores registrados deberían poder ver las encuestas no publicadas, pero los demás usuarios no. Una vez más: cualquier funcionalidad que necesites agregar debe estar acompañada por pruebas o 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 de ellos. Van a seguir cumpliendo su función mientras uno continúa desarrollando su programa.

Algunas veces los tests van a necesitar actualizarse. Supongamos que corregimos nuestras vistas para que solamente se publiquen preguntas con opciones. En ese caso, muchos de nuestros tests existentes van a fallar - *diciéndonos qué algunos test se necesitan actualizar y corregir*, así que hasta cierto punto los tests pueden cuidarse a sí 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 volver inmanejables. Algunas buenas prácticas son las siguientes:

- Separa la clase TestClass para cada modelo o vista
- Separa los métodos de test para cada conjunto de condiciones a comprobar
- Usa nombres en los métodos de pruebas que describan su función

Pruebas adicionales

Este tutorial únicamente introduce algunos conceptos básicos sobre pruebas. Hay bastante más cosas que se puede hacer y existen herramientas muy útiles a tu disposición para lograr cosas mucho más interesantes.

Por ejemplo, mientras que nuestros tests han cubierto la lógica interna de un modelo y la forma en que nuestras vistas publican información, podríamos usar un framework “in-browser” como Selenium (http://www.seleniumhq.org/) para testear la manera en que el HTML se renderiza en un navegador. Estas herramientas nos permiten no sólo verificar el comportamiento de nuestro código Django, sino también por ejemplo, nuestro JavaScript. Es algo muy curioso ver ejecutar los tests en un navegador y empezar a interactuar con nuestro sitio como si un humano lo estuviera controlando! Django incluye una clase LiveServerTestCase para facilitar la integración con herramientas como Selenium.

Si tenemos una aplicación compleja, podríamos querer correr los tests automáticamente con cada commit con el propósito de integración continua, de tal manera que podemos automatizar - al menos parcialmente - el control de calidad.

Una buena forma de encontrar partes de tu aplicación sin testear es verificar la cobertura del código. Esto también ayuda a identificar código frágil o muerto. Si no podemos testear un fragmento de código, en general significa que el código debe refactorizarse o borrarse. La herramienta Coverage (https://pypi.python.org/pypi/coverage) puede ayudarte a identificar código muerto.

¿Qué sigue?

Cuando te sientas confortable creando tests en Django, puedes leer la sexta parte de este tutorial para aprender a manejar archivos estáticos en Django.

CAPÍTULO 08

Escribe tu primera aplicación Django, parte 06

Continuamos con la sexta parte de este tutorial. Una vez que hemos testeado nuestra aplicación de encuestas Web, vamos a agregar una hojas de estilo y una imagen.

Aparte del código HTML generado por el servidor, las aplicaciones web generalmente necesitan servir archivos adicionales - tales como imágenes, JavaScript o CSS - estos archivos son necesarios para representar páginas Web completas. En Django, nos referimos a estos archivos como "archivos estáticos" (static files).

Para proyectos pequeños, esto no es un gran problema, ya que sólo necesitas mantener los archivos estáticos en algún lugar donde el servidor Web pueda encontrarlos. Sin embargo, en proyectos grandes - especialmente aquellos que comprenden múltiples aplicaciones - al tratar con varios conjuntos de archivos estáticos proporcionados por cada aplicación esto puede ser muy complicado.

Esto es por lo que se creó el paquete django.contrib.staticfiles, que se encarga de recoger los archivos estáticos de cada una de tus aplicaciones Django (y cualquier otro lugar que especifiques) en una ubicación única, para que pueda servirse fácilmente en producción.

Personaliza la apariencia de tu aplicación

En primer lugar, crea un directorio llamado `státic` en el directorio encuestas. Django buscará los archivos estáticos allí, de manera similar a cómo Django encuentra las plantillas dentro de encuestas/templates/.

El archivo de configuración settings.py, contiene una variable; STATICFILES_FINDERS que contiene una lista de los lugares donde Django busca los diversos archivos estáticos. Uno de los valores por defecto es AppDirectoriesFinder que busca un subdirectorio "státic" en cada uno de las aplicaciones registradas en INSTALLED_APPS, como el de encuestas que acabamos de crear. El sitio administrativo utiliza la misma estructura de directorios para los archivos estáticos.

Dentro del directorio static que acabas de crear, crea otro directorio llamado encuestas y dentro de este crea un archivo llamado style.css. Es decir la hoja de estilos; debe estar en el directorio de encuestas así: /templates/static/encuestas/style.css. Esta es la forma en cómo funciona el AppDirectoriesFinder el buscador de archivos estáticos, podemos hacer referencia a este archivo estático en Django simplemente como encuestas/style.css de forma similar a como nos referimos a la ruta de acceso a las plantillas.

Espacios de nombres para archivos estáticos

Al igual que las plantillas, podrías poner todos tus archivos estáticos directamente en encuestas/static (en lugar de crear otro subdirectorio encuestas), pero en realidad

sería una mala idea. Django elegirá el primer archivo estático que encuentra y que coincida con el nombre pedido y si tienes un archivo estático con el mismo nombre, pero de una *diferente* aplicación, Django no podría distinguir entre ellos. Debemos asegurarnos de que Django pueda encontrarlos y la mejor manera de que esto ocurra es usando *espacios de nombres*. Es decir, poniendo los archivos estáticos en el interior de *otro directorio* con el nombre de la propia aplicación.

Coloca el siguiente código en la hoja de estilos en encuestas/static/encuestas/style.css:

encuestas/static/encuestas/style.css

```
li a {
    color: green;
}
```

A continuación, agrega lo siguiente en la parte superior de la plantilla: encuestas/templates/encuestas/index.html:

encuestas/templates/encuestas/index.html

```
{% load staticfiles %}

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

La etiqueta `{% load staticfiles %}` carga todos los archivos estáticos de la librería de plantillas staticfiles. Las etiqueta de plantilla {% static %} usan la URL absoluta de los en la que se encuentran los archivos estáticos.

Eso es todo lo que necesitas hacer en desarrollo, para utilizar hojas de estilo. Recarga la página http://localhost:8000/encuestas/ y podrás ver que cada vínculo de una pregunta es de color verde (estilo Django!) lo que significa que la hoja de estilos se ha cargado correctamente.

Agrega una imagen de fondo

A continuación, vamos a crear un subdirectorio para imágenes. Crea un subdirectorio llamado: imagenes en el directorio encuestas/static/encuestas/. Dentro de este directorio, pon una imagen llamada fondo.gif. En otras palabras, pon la imagen en el directorio encuestas/static/encuestas/imagenes/fondo.gif.

A continuación, añade a la hoja de estilos (encuestas/static/encuestas/style.css), lo siguiente:

encuestas/static/encuestas/style.css

```
body {
   background: white url("imagenes/fondo.gif") no-repeat right
                              bottom;
}
```

Actualiza la página http://localhost:8000/encuestas/ y deberías ver colocada la imagen, en el fondo en la parte inferior derecha de la pantalla.

Por supuesto que, las etiqueta de plantilla {% static %} no estás disponible para su uso en los archivos estáticos, tal como la hoja de estilos que no son generadas por Django. Siempre debes usar **rutas relativas** para vincular los archivos estáticos entre otras cosas porque entonces puedes cambiar la variable STATIC_URL (utilizado por la etiqueta de plantilla static para generar las URL) sin tener que modificar un montón de rutas en los archivos estáticos también.

Estos son los **fundamentos básicos** para obtener más información sobre la configuración y uso de archivos estáticos, consulta la guía oficial, en la que se discuten sobre archivos estáticos servidos en producción.

¿Qué sigue?

El tutorial de inicio para principiantes termina aquí, por el momento. Mientras tanto, es posible que quieras consultar algunas otros temas, puedes consultar la documentación oficial disponible en *https://docs.djangoproject.com/*.

Si está familiarizado con paquetes Python y estás interesado en aprender cómo convertir tu aplicación de encuestas en una "aplicación reutilizable", échale un vistazo al siguiente tema ¿Cómo escribir aplicaciones reutilizables?

CAPÍTULO 09

Tutorial avanzado: Como escribir aplicaciones reusables

Continuamos con la séptima parte de este tutorial. En este capítulo vamos a convertir nuestra aplicación de encuestas Web en un paquete independiente Python de manera que pueda rehusarse en nuevos proyectos y compartirse con otras personas.

Si todavía no has completado los tutoriales del 1 al 6, te animamos a que lo revises, para que tu proyecto de ejemplo coincida con el que se describe a continuación.

Material reusable

Cuesta mucho trabajo para diseñar, construir, probar y para mantener una aplicación Web. Muchos de estos problemas son comunes en los proyectos Python y Django. No sería genial si ¿pudiéramos ahorrar algo de este trabajo repetido y después volver a usarlo?

La reusabilidad es la forma de vida en python. The Python Package Index (https://pypi.python.org/pypi) contiene un extenso número de paquetes que puedes utilizar en tus propios programas de Python. Django en sí mismo es un paquete Python. Esto significa que puedes tomar un paquete Python existente o una aplicación Django y crear tu propio proyecto Web. Solamente necesitaras escribir algunas partes para crear un proyecto único.

Digamos que necesitamos comenzar un nuevo proyecto, que necesita una aplicación de Encuestas como en la que hemos estado trabajando. ¿Cómo hacemos esta aplicación reutilizable? Afortunadamente, estamos en el camino correcto Hemos empezamos por desacoplar la aplicación de Encuestas del proyecto en la URLconf en el t*utorial 3*, mediante el uso de include. En este tutorial, vamos a tomar otras medidas para hacer la aplicación más fácil de usar en nuevos proyectos y prepararla para publicar para que otros puedan usarla e instalarla.

¿Paquete? ¿Aplicación?

Un paquete en Python (http://docs.python.org/tutorial/modules.html#packages) proporciona una manera de agrupar código relacionado entre sí, para que la reutilización sea más fácil. Un paquete puede contener uno o más archivos de código python (también conocidos como “módulos”).

Un **paquete** puede ser importado con la sentencia import foo.bar o from foo import bar. Para que un directorio (como el de Encuestas) pueda ser considerado un paquete, debe contener un fichero especial llamado **__init__.py,** incluso si este fichero está vacío.

Un **aplicación** de Django es justamente un paquete Python que esta específicamente diseñado para utilizare en un proyecto de Django. Un aplicación puede utilizar las convenciones comunes de Django, por ejemplo tener un archivo models.py.

Utilizamos el termino empaquetar para describir el proceso de crear un paquete Python mas fácil de instalar por otros. Esto puede parecer un poco confuso al principio, nosotros lo sabemos.

Completando una aplicación reutilizable

Después de completar los tutoriales anteriores, tu proyecto se debe parecer a este:

```
misitio/
    manage.py
    misitio /
        __init__.py
        settings.py
        urls.py
        wsgi.py
    encuestas/
        __init__.py
        admin.py
        migrations/
            __init__.py
            0001_initial.py
        models.py
        static/
            encuestas/
                imagenes/
                    fondo.gif
                style.css
        templates/
            encuestas/
                detalles.html
                index.html
                resultados.html
        tests.py
        urls.py
        views.py
    templates/
        admin/
            base_site.html
```

Creaste un directorio llamado misitio/templates en el t*utorial 2* y otro llamado encuestas/templates en el *tutorial 3*. Ahora tal vez es más claro por qué se optó por tener directorios de plantillas separadas para el proyecto y para la aplicación: todo lo que forma parte de la aplicación se encuentra en el directorio encuestas. Esto hace que la aplicación sea independiente y más fácil de usar en un nuevo proyecto.

El directorio de encuestas, ahora se puede copiar en un nuevo proyecto Django e inmediatamente utilizarlo. No está listo aun para ser publicado, para eso es necesario primero empaquetar la aplicación, para hacer más fácil para que otros lo instalen.

Instalación de algunos requisitos previos

El estado actual de los paquetes Python es un poco confuso, ya que se cuentan con varias herramientas para crear paquetes, para este tutorial, vamos a utilizar **`setuptools`** (https://pypi.python.org/pypi/setuptools) para construir nuestro paquete. Este es uno de los proyectos mantenidos por la mayor de las comunidades encargados del proyecto **`distribute.`** También vamos a usar **`pip`** (https://pypi.python.org/pypi/pip) para instalarlo y desinstalarlo después de que hayamos terminado. Debes instalar estos dos paquetes ahora. Si necesitas ayuda, puedes consultar la guía de referencia: como instalar Django usando pip. Puedes instalar setuptools de la misma manera.

Empaquetando una aplicación

Los *paquetes* Python se refieren a preparar una aplicación en un formato específico, para que puedan ser fácilmente instalados y usados por otros usuarios. Django en sí mismo es un paquete como este, solo que es más grande. Para una pequeña aplicación de encuestas, este proceso no es demasiado complicado.

1. En primer lugar, crea directorio padre para la aplicación Encuestas, fuera del proyecto Django, a este directorio lo llamaremos django-encuestas.

El nombre de tu aplicación

Al cambiar o al elegir un nombre para tu paquete, es necesario comprobar el nombre de tu paquete con los nombres de los paquetes y recursos en PyPI , para evitar conflictos con los paquetes existentes. A menudo es útil anteponer la palabra django- al nombre del módulo, al crear un paquete de distribución. Esto ayuda a los usuarios que buscan aplicaciones Django identificar alguna aplicación en específico.

---

2. Muévete al directorio encuestas, dentro del directorio de django-encuestas.

3. Crea un archivo django-encuestas/README.txt con el contenido siguiente, en formato rst:

`django-encuestas/README.rst`

```
=========
Encuestas
=========

Encuestas es una simple aplicación Django, de encuestas
Web. Para cada pregunta, el visitante puede elegir entre un
número fijo de respuestas u opciones.
```

```
La documentación detallada está en el directorio "doc".

Inicio rápido
-------------

1. Agrega la aplicación ``encuestas`` a tu archivo de
configuración   ``settings.py``, en la variable
``INSTALLED_APPS`` asi::

     INSTALLED_APPS = (
         ...
         "encuestas",
     )

2. Incluye la aplicación de encuestas en la URLconf de tu
proyecto   urls.py de este modo::

     (r'^encuestas/', include('encuestas.urls')),

3. Ejecuta el comando `python manage.py migrate` para crear
los   modelos para las encuestas.

4. Inicia el servidor de desarrollo y visita la página
http://127.0.0.1:8000/admin/      para crear una encuesta
(Necesitas habilitar la interfaz administrativa     para
poder crear encuestas).

5. Visita la página http://127.0.0.1:8000/encuestas/ para
   participar en una encuesta.
```

4. Crea un archivo django-encuestas/LICENSE La elección de una licencia va más allá del alcance de este tutorial, pero basta con decirte que el código liberado sin una licencia es *inútil*. Django y muchas de las aplicaciones compatibles con Django son distribuye bajo la licencia BSD, sin embargo, eres libre de elegir tu propia licencia. Sólo ten en cuenta que la elección de unas licencias afectará a quien pueda usar tu código.

5. A continuación crea un archivo llamado **`setup.py`** que proporciona información detallada sobre cómo construir e instalar la aplicación. Una explicación completa de este archivo va más allá del alcance de este tutorial, pero en http://www.pythonhosted.org/setuptools/setuptools.html tiene una excelente explicación para usar setuptools.

Crea un archivo django-encuestas/setup.py con el siguiente contenido:

django-encuestas/setup.py

```
import os
from setuptools import setup

with open(os.path.join(os.path.dirname(__file__),
    'README.rst')) as readme:
        README = readme.read()

# Permite a setup ejecutarse desde cualquier ruta.
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__)
    , os.pardir)))
```

```
setup(
    name='django-encuestas',
    version='0.1',
    packages=['encuestas'],
    include_package_data=True,
    license='BSD License',  # Ejemplo de licencia
    description='Una simple aplicación Django de encuestas.',
    long_description=README,
    url='http://www.example.com/',
    author='tu nombre',
    author_email='yourname@example.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
        'Intended Audience :: Developers',
        # Ejemplo de licencias
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        # Remplázalo con la versión de python adecuada.
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.2',
        'Programming Language :: Python :: 3.3',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)
```

6. Sólo los módulos y paquetes Python se incluyen en el paquete por defecto. Para incluir archivos adicionales, tendrás que crear un archivo MANIFEST.in la documentación sobre **`setuptools`** a la que nos referimos en el paso anterior trata estos detalles más a fondo. Para incluir las plantillas, los archivos README.rst y la LICENCE, necesitas crear un archivo django-encuestas/MANIFEST.in con el contenido siguiente:

django-encuestas/MANIFEST.in

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

Ten en cuenta que el directorio docs no se incluirá en el paquete a menos que agregues algunos archivos al mismo. Muchas aplicaciones Django también proporcionan la documentación en línea a través de sitios como readthedocs.org (https://readthedocs.org/).

7. Luego construimos el paquete con el comando `python setup.py sdist` (ejecuta el comando desde dentro de la carpeta django-encuestas). Este comando crea un directorio llamado `dist` y construye un nuevo paquete comprimido, django-encuestas-0.1.tar.gz.

Para más información sobre el empaquetado de aplicaciones, puedes consultar en línea: https://packaging.python.org/en/latest/distributing.html para más información sobre el empaquetado y distribución de paquetes Python.

Usar nuestra aplicación

Puesto que movimos el directorio encuestas, fuera del directorio de proyecto, este ya no está funcionando. Ahora repararemos eso instalando una nueva aplicación el paquete django-encuestas que hemos creado.

Instalando una librería como un usuario del sistema.

En los siguientes pasos instalaremos la aplicación django-encuestas como una librería de usuario, este tipo de instalación tienen un montón de ventajas con respecto a la instalación del paquete en el sistema, tal como se utiliza en sistemas en los que no se tiene acceso como administrador, así como para prevenir que el paquete pueda afectar otros servicios o a otros usuarios.

Ten en cuenta que las instalaciones por usuario pueden afectar al comportamiento del sistema, de las herramientas que se ejecutan como ese usuario, por lo que usar **`virtualenv`** es una solución más robusta (ver más abajo).

---

1. Para instalar el paquete usa `pip` (¿ya lo instalaste, cierto?)

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

2. Con suerte, nuestro proyecto Django debe trabajar correctamente otra vez. Ejecuta el servidor para confirmarlo.

3. Para desinstalar el paquete, usa pip:

```
pip uninstall django-encuestas
```

Publica tu aplicación

Ahora que hemos empaquetado y testeado nuestro paquete django-encuestas, estamos listos para compartirlo con ¡todo el mundo! Si esto no fuera un simple ejemplo, ahora podríamos:

- Enviar el paquete por correo electrónico a un amigo.
- Subir el paquete a nuestro sitio Web.
- Postear el paquete a un repositorio público, tal como The Python Package Index (https://pypi.python.org/pypi) o a packaging.python.org.

Instalar paquetes Python con virtualenv

Anteriormente, instalamos la aplicación de encuestas como una biblioteca de usuario. Esta tiene algunas desventajas:

- Al modificar las bibliotecas del sistema podemos afectar otro software Python del sistema.
- No podremos hacer funcionar múltiples versiones de este paquete (u otras con el mismo nombre).

Típicamente, estas situaciones únicamente se presentan si estas manteniendo varios proyectos Django al mismo tiempo. Si ese es tu caso, la mejor solución es utilizar *virtualenv*.

Esta herramienta te permite mantener un ambiente aislado, usando múltiples paquetes Python, cada uno con sus propias copias de librerías y nombres de paquetes. Puedes aprender más visitando el sitio Web de virtualenv (http://www.virtualenv.org/).

¿Qué sigue?

En el siguiente capítulo te mostraremos como contribuir con el código fuente de Django.

CAPÍTULO 10

Escribe tu primer parche para Django

Introducción

¿Interesado en devolver a la comunidad un poco?, tal vez encontraste un error en Django que te gustaría corregir, o tal vez hay una pequeña característica que te gustaría añadir al código.

Contribuir con el código fuente es en sí mismo, la mejor manera de abordar y ver nuestras propias preocupaciones retribuidas en Django. Esto puede parecer un poco intimidante al principio, pero en realidad es bastante simple. Esta guía te llevara paso a paso, a través de todo el proceso, así que puedes aprender por medio de este ejemplo.

¿A quién va dirigido este tutorial?

Para este tutorial, es de esperarse que tengas por lo menos un conocimiento básico de la forma en que funciona Django. Esto quiere decir que debes ir poco a poco aprendiendo de una forma cómoda sobre los tutoriales existentes. Además, se necesita tener una buena comprensión de Python. Pero si no la tienes no te preocupes puedes leer "Inmersión en python" es un libro fantásticos en línea (y gratis) para comenzar a programar en Python.

Si es que aun no estás familiarizado con los sistemas de control de versiones tipo **Trac, Git**, encontrarás que en este tutorial se incluyen vínculos con suficiente información para comenzar. Sin embargo, es probable que quieras leer un poco más acerca de estas herramientas diferentes entre sí, si es que vas a contribuir a Django con regularidad.

En su mayor parte, este tutorial trata de explicar tanto como sea posible, de modo que pueda ser de utilidad a audiencias más amplias.

Dónde obtener ayuda:

Si tienes problemas a lo largo de este tutorial, por favor postea un mensaje a django-developers o a través del chat #django-dev en irc.freenode.net para chatear con otros usuarios de Django que podrán resolver tus dudas y ayudarte.

¿Qué cubre este tutorial?

A lo largo de este tutorial te mostraremos la forma de contribuir con un parche a Django por primera vez. Al final de este tutorial, tendrás un conocimiento básico tanto de las herramientas, como de los procesos involucrados. En concreto, vamos a cubrir los siguientes temas:

- La instalación de Git.
- Cómo descargar una copia de desarrollo de Django.
- Como ejecutar un paquete de pruebas en Django.
- Como escribir una prueba para un parche.
- Como escribir el código para un parche.
- Testear un parche.
- Generar un archivo de revisión de cambios.
- ¿Dónde encontrar más información?

Una vez que hayas terminado con el tutorial, puedes buscar a través del resto de la documentación oficial, contiene una gran cantidad de información y es una lectura obligada para cualquier persona que quiera convertirse en un colaborador habitual de Django. Si tienes alguna pregunta, es probablemente que allí esté la respuesta.

Instalar Git

Para este tutorial, necesitas instalar Git, (http://www.git-scm.com/download) para descargar la versión actual de desarrollo de Django y generar archivos de revisión para posteriormente hacer los cambios.

Para comprobar si tienes o no instalado Git, introduce el siguiente comando en la terminal asi: **`git.`** Si recibes un mensaje diciendo que el comando no puede ser encontrado, tendrás primero que descargarlo e instalarlo.

Si no estás familiarizado con Git, puedes encontrar información útil sobre los comandos (una vez instalado) escribiendo **`git help`** en la línea de comandos.

Como obtener una copia de la versión de desarrollo de Django

El primer paso para contribuir a Django es obtener una copia del código fuente. Desde la línea de comandos, utiliza el comando **`cd`** para navegar hasta el directorio donde quieras almacenar una copia local de Django.

Descarga el código fuente desde el repositorio de Django con el siguiente comando:

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

Nota: Para los usuarios que deseen usar **`virtualenv`** (http://www.virtualenv.org/) pueden utilizar:

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

(Donde django es el directorio donde hemos clonado el repositorio, que contiene el archivo setup.py) Para así enlazar el directorio clonado con el entorno virtual. Esta es una excelente opción, para aislar la copia de desarrollo de Django del resto de el sistema para evitar posibles conflictos entre paquetes.

Retornar a una revisión previa de Django

Para este tutorial vamos a utilizar el ticket #17549: https://code.djangoproject.com/ticket/17549 como caso de estudio, por lo que vamos a retroceder en el historial de versiónes de git sobre Django, antes de de aplicar el ticket para el parche. Esto nos permitirá cubrir todos los pasos necesarios para escribir ese parche desde cero, incluyendo la ejecución de un conjunto de test's o pruebas en Django.

Ten en cuenta que, vamos a estar utilizando una versión antigua del tronco de Django para los efectos de este tutorial, ¡siempre se debe utilizar la revisión actual de desarrollo de Django cuando se trabaja en un parche para un ticket!

Nota: El parche para esta entrada fue escrita por Ulrich Petri, y se aplicó a Django como commit ac2052ebc84c45709ab5f0f25e685bf656ce79bc. Por lo tanto, vamos a aprovechar la revisión de Django justo antes de aplicar el commit 39f5bc7fc3a4bb43ed8a1358b17fe0521a1a63ac.

Navega al directorio raíz de Django (que es el que contiene django, docs, tests, AUTHORS, etc.). A continuación, puedes checar la revision de Django que vamos a utilizar a continuación en el tutorial:

```
git checkout 39f5bc7fc3a4bb43ed8a1358b17fe0521a1a63ac
```

Ejecutando la suite de pruebas en Django por primera vez

Cuando contribuimos a Django es muy importante que los cambios en el código no introduzcan más errores en otras áreas de Django. Una forma de comprobar que Django trabaja bien después de realizar algún cambio, es mediante la ejecución de una suite de pruebas. Si todas las pruebas pasan, entonces se puede estar razonablemente seguro de que los cambios no han roto completamente a Django. Si nunca has ejecutado una suite de pruebas en Django antes, es una buena idea

ejecutarla una vez antes, sólo para familiarizarse con la forma en que se comporta y con la forma de salida.

Puedes ejecutar el conjunto de pruebas, simplemente situándote con el comando cd en el directorio tests/, si estás usando GNU/Linux, Mac OS X o algún otro sabor de Unix, puedes ejecutar:

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

Si estás en Windows, lo anterior debería funcionar siempre y cuando estés utilizando "Git Bash" que es proporcionado por la instalación Git por defecto. GitHub tiene un tutorial muy bueno en: https://help.github.com/articles/set-up-git#platform-windows

---

Nota: Si utilizas virtualenv, puede omitir PYTHONPATH=.. Cuando ejecutes las pruebas. Esto indica a Python que busque a Django en el directorio padre de pruebas tests. virtualenv pone su copia de Django en el PYTHONPATH automáticamente.

Ahora siéntate y relájate un poco. Todo el conjunto de pruebas de Django consiste en más de 4800 diferentes test`s, por lo que pueden tardarse en ejecutarse entre 5 y 15 en función de la velocidad de tu ordenador.

Mientras la suite de pruebas de Django se está ejecutando, verás una secuencia de caracteres que representan el estado de cada prueba al ser ejecutada. E indica que un error se ha lanzado durante una prueba y F indica que una de las pruebas de aserción ha fallado. Ambos se consideran fallos de prueba. Mientras tanto, x y s indican fallas esperadas y las pruebas omitidas, respectivamente. Los puntos indican que se pasaron las pruebas.

Las pruebas que suelen ser omitidas son debido a que faltan bibliotecas externas necesarias para ejecutar la prueba, puedes consultar la documentación oficial para obtener una lista de todas las dependencias necesarias y asegurarte de instalar cualquiera de las bibliotecas faltantes para las pruebas relacionadas con los cambios que se está haciendo (que no serán necesaria para este tutorial).

Una vez que se completan las pruebas, recibirás un mensaje informándote, si el conjunto de pruebas aprobó o no. Dado que aún no se ha realizado ningún cambio al código de Django, la prueba de toda la suite **debería** aprobar. Si obtienes fallas o errores asegúrate de que has seguido todos los pasos anteriores correctamente.

Ten en cuenta que el tronco (trunk) de Django puede no ser estable siempre. Cuando desarrollamos en el tronco, podemos comprobar y determinar si los errores son específicos de la máquina en que estamos trabajando o si también están presenten en las versiones oficiales de Django. Si das clic en una vista en particular, se puede ver la "Matriz de configuración", que muestra las fallas desglosadas por el tipo de versión de Python y la base de datos back-end usada.

Nota:

Para este tutorial y ticket en el que estamos trabajando, las pruebas son ejecutadas usando como base de datos SQLite, sin embargo, es posible (y, a veces es necesario) ejecutar las pruebas usando otra u otras base de datos.

Escribir algunas pruebas para el ticket

En la mayoría de los casos, para ser aceptado un parche en Django este tiene que incluir pruebas. Para parches de corrección de errores, esto significa escribir una prueba de regresión para asegurar que el error no vuelva a ser reintroducido en Django más adelante. Una prueba de regresión debería ser escrita de tal manera que produzca un error, mientras que todavía existe el error y pasar una vez que el error se halla corregido. Para los parches con nuevas características, deberán ser incluidas pruebas que garanticen que las nuevas características funcionan correctamente. Estas también deberían fallar cuando la nueva característica no está presente y luego pasar una vez que se haya aplicado el parche.

Una buena manera de hacer esto es escribir las nuevas pruebas en primer lugar, antes de hacer cualquier cambio en el código. Este estilo de desarrollo es llamado "test-driven development" y se puede aplicar tanto en proyectos completos y en parches individuales.

Después de escribir las pruebas, a continuación las ejecutamos para asegurarnos que de no se han hecho cambios (ya que aun no se ha corregido este error o añadido todavía esa característica). Si las nuevas pruebas no fallan, tendrás que arreglarlas para que lo hagan.

Después de todo, un análisis de regresión que pasa sin importar si un error está presente o no, no es muy útil en la prevención de estos errores si se repitan en el futuro.

Ahora, para nuestro ejemplo práctico pongamos manos a la obra:

Escribiendo algunas pruebas para el ticket #17549

El Ticket #17549 describe como agregar una pequeña característica, de la siguiente forma:

> Es útil para un campo URLField para darle la opción de abrir la dirección URL, de lo contrario se podría utilizar de igual forma un CharField.

Con el fin de resolver este ticket, vamos a añadir un método render para el widget AdminURLFieldWidget con el fin de mostrar un vínculo al hacer clic encima de la entrada del widget. Sin embargo antes de realizar estos cambios, vamos a escribir un par de pruebas para comprobar que las modificaciones que hicimos funcionan correctamente y lo continuaran haciendo en el futuro.

Navega hasta el directorio de Django tests/regressiontests/admin_widgets/ y abre el archivo test.py. Agrega el siguiente código justo en la línea 269 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:<input class="vURLField" name="test"
            type="text" value="http://example.com" /></p>'
        )

    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.com</a><br
            />Change:<input class="vURLField" name="test"
            type="text" value="http://example-äüö.com" /></p>'
        )

    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%20tex
                t%3C/sometag%3E">http://example.com/&lt;sometag&
                gt;some text&lt;/sometag&gt;</a><br
                />Change:<input class="vURLField" name="test"
                type="text"
                value="http://example.com/<sometag>some
                text</sometag>" /></p>'
        )
        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%20text%3C/sometag%
                3E">http://example-äüö.com/&lt;sometag&gt;some
                text&lt;/sometag&gt;</a><br />Change:<input
                class="vURLField" name="test" type="text"
                value="http://example-äüö.com/<sometag>some
                text</sometag>" /></p>'
        )
```

Las nuevas pruebas verifican que el método render agrega correctamente los nuevos trabajos en un par de situaciones diferentes.

Pero esta cosa de los test's, parece un tanto difícil...

Si nunca has tenido que lidiar con los tests, puede parecer un poco duro escribir una prueba o un test a primera vista. Afortunadamente, las pruebas son un tema *muy* recurrente en programación de ordenadores, así que hay mucha información útil disponible por ahí:

- Para escribir pruebas en Django, como primer vistazo se puede encontrar en la misma documentación en como "testear aplicaciones en Django".
- Inmersión en Python (un libro gratis en línea para desarrolladores principiantes de Python) incluye un gran introducción a pruebas unitarias.

- Después de leer esto, si todavía quieres algo un poco más sustancioso puedes hundirle el diente a la documentación de Python, en especial a la sección dedicada a pruebas o tests, busca en: http://docs.python.org/library/unittest.html

Ejecuta la nueva prueba

Recuerda que todavía, no se han realizado modificaciones a `AdminURLFieldWidget`, así que nuestras pruebas van a fallar. Vamos a correr todo el conjunto de pruebas en la carpeta `model_forms_regress` para asegurarnos de que es lo que realmente sucede. Desde la línea de comandos, cámbiate de directorio con `cd` ala carpeta `tests/` y ejecuta:

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

Si las pruebas se han ejecutado correctamente, deberías ver tres fallos correspondientes a cada uno de los métodos de prueba que hemos añadido. Si todas las pruebas pasan, entonces debes asegurarte de que no haber agregado las nuevas pruebas en otro directorio.

Escribe el código para tu ticket

A continuación vamos a añadir la funcionalidad descrita en el ticket #17549 a Django.

Escribir el código para el ticket #17549

Navega hasta el directorio `django/django/contrib/admin/` y abre el archivo widgets.py. Busca la clase `AdminURLFieldWidget` y añade en la línea 302, el siguiente método `render` después de el método `__init__` existente.

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

Verifica que la prueba pase ahora

Una vez que hayas terminado las modificaciones de Django, necesitas asegurarte que las pruebas que escribimos antes pasen correctamente, para que podamos ver si el código que escribimos arriba está trabajando correctamente. Para ejecutar las pruebas cámbiate a la carpeta admin_widgets con el comando cd (o como prefieras) y sitúate en el directorio tests/ y ejecuta la prueba

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

¡Oops, menos mal que escribimos esta prueba! Aún tenemos 3 fallos con la excepción siguiente:

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

Nos hemos olvidado de importar el método smart_urlquote. Seguimos adelante y añadimos el método de importación smart_urlquote al final de la línea 13 del archivo django/contrib/admin/widgets.py de la siguiente forma:

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

Vuelve a ejecutar las pruebas y debería pasarlas todas. Si no lo hace, asegúrate de haber modificado correctamente la clase AdminURLFieldWidget como se muestra arriba y de haber copiado las nuevas pruebas correctamente.

Ejecuta el paquete de pruebas Django por segunda vez

Una vez que hayas verificado que el parche y la prueba están funcionando correctamente, es una buena idea ejecutar toda la suite de pruebas de Django sólo para comprobar que el cambio no ha introducido ningún otro error en otras áreas de Django. Aunque paso con éxito todo el conjunto de pruebas esto no garantiza que el código esté libre de errores, lo que hace es ayudar a identificar muchos errores y regresiones que de otro modo podrían pasar desapercibidas.

Para ejecutar la totalidad del conjunto de pruebas de Django nos cambiamos al directorio tests/ con cd y ejecutamos

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

Siempre y cuando no se vea ningún fallo, esto está bien por el momento. Ten en cuenta que esta revisión También hizo un pequeño cambio al dar formato al widget small CSS. Puedes hacer el cambiar si lo deseas, pero lo vamos a omitir por el momento, en aras de la brevedad.

Escribe la documentación

Esta es una nueva característica, por lo que debe ser documentada. Agrega lo siguiente en la línea 925 de django/docs/ref/models/fields.txt debajo de la actual documentación para URLField:

```
.. versionadded:: 1.5

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

Para obtener más información sobre escribir documentación, consulta la guía oficial, en esta se incluye una explicación acerca de cómo construir una copia de la documentación a nivel local, para poder tener una vista previa del HTML que se generará.

Genera un parche para tus cambios

Ahora es el momento para generar un archivo de revisión que se pueden cargar en Trac o pueda ser aplicado a otra copia de Django. Para echar un vistazo a los contenidos de el parche, ejecutamos el siguiente comando:

```
git diff
```

Este comando mostrará las diferencias entre la copia actual de Django (con los cambios) y la revisión que checamos inicialmente a principios de este tutorial (sin cambios).

Una vez que hayas terminado de ver el parche, pulse la tecla q para salir de nuevo a la línea de comandos. Si el contenido del parche es correcto, puedes ejecutar el siguiente comando para guardar el archivo de parche en el directorio de trabajo actual

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

Ahora debes tener un archivo en el directorio raíz de Django llamado 17549.diff. Este parche contiene todos los cambios que has hecho y debe verse así:

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

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

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

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

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

 Relationship fields
 ===================
```

```
diff --git a/tests/regressiontests/admin_widgets/tests.py
b/tests/regressiontests/admin_widgets/tests.py
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="vDateField" name="test_0"
size="10" /><br />Zeit: <input value="09:30:00" type="text"
class="vTimeField" name="test_1" size="8" /></p>',
                 )

+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:<input class="vURLField" name="test" type="text"
value="http://example.com" /></p>'
+        )
+
+    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.com</a><br
/>Change:<input class="vURLField" name="test" type="text"
value="http://example-äüö.com" /></p>'
+        )
+
+    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%3E"
>http://example.com/&lt;sometag&gt;some
text&lt;/sometag&gt;</a><br />Change:<input class="vURLField"
name="test" type="text" value="http://example.com/<sometag>some
text</sometag>" /></p>'
+        )
```

```
+        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%20text%3C/sometag%3E">http://examp
le-äüö.com/&lt;sometag&gt;some text&lt;/sometag&gt;</a><br
/>Change:<input class="vURLField" name="test" type="text"
value="http://example-äüö.com/<sometag>some text</sometag>"
/></p>'
+        )
```

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

A sí que, ¿ahora que sigue?

¡Felicidades, has generado tu primer parche para Django! Ahora que has aprendido como hacerlo, puedes usar estas habilidades para ayudar a mejorar la base del código fuente de Django. La generación de parches y tickets para adjuntarlos a Trac es útil, sin embargo, ya que estamos usando git - recomendado como versión de control, vamos a usarlo.

Ya que nunca hemos subido (committed) nuestros cambios a nivel local, necesitamos hacer lo siguiente para obtener una nueva rama de git como punto de partida

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

Más información para los nuevos contribuyentes

Antes de que te pongas a escribir parches para Django, hay un poco más de información sobre las contribuciones que probablemente deberías echarles un vistazo:

- Debes asegurarse de leer la documentación de Django sobre como subir tickets y parches. Cubre temas como etiquetas en Trac, estilo de codificación para parches, y muchos detalles importantes.
- Si eres un contribuyente nuevo, deberías leer la documentación sobre “La documentación para primeros contribuyentes” Contiene un montón de buenos consejos y prácticas para principiantes que quieren ayudar con Django.

- Después de todo esto, si todavía tienes ganas de obtener más información acerca de la forma de contribuir a Django, siempre puedes navegar por el resto de la documentación oficial. Contiene un montón de información útil y debería ser tu primera fuente de información para responder a cualquier pregunta que te pueda surgir.

Encuentra tu primer Ticket real

Una vez revisada alguna de esta información, puedes estar listos para salir y encontrar un ticket por tu propia cuenta y para escribir un parche para este. Presta especial atención a las entradas marcadas con el criterio "easy pickings". Estos tickets son a menudo mucho más simples en la naturaleza y son ideales para los contribuyentes noveles. Una vez que estés familiarizado con la forma de contribuir a Django, puedes pasar a escribir parches para entradas más difíciles y complicadas.

Si lo que desea es empezar ya (¡y nadie te culparía de ello!), Prueba echar un vistazo a la lista de easy tickets that need patches (tickets que necesitan parches) y a la de easy tickets that have patches which need improvement. Si está familiarizado con la escritura de pruebas, también puede consultar la lista de easy tickets that need tests. Sólo recuerda seguir las directrices sobre entradas que se mencionan en el enlace a la documentación de Django en sobre como subir tickets y parches.

¿Qué sigue?

Después de que un ticket tiene un parche, tiene que ser revisado por un segundo par de de ojos. Después de cargar un parche o de que se presente una solicitud de extracción, asegurarte de actualizar los metadatos del ticket mediante el establecimiento de las banderas correspondientes a el ticket que diga "has patch", "doesn't need tests", etc. ("tiene parche", "no necesita pruebas)", para que otros puedan encontrarlos para su revisión. Contribuyendo no necesariamente significa escribir un parche desde cero. La revisión de parches existentes es también una contribución muy útil.

Licencia y Copyrigth

Licencia de documentación libre de GNU

Versión 1.2, November 2002

Esta es una traducción no oficial de la GNU Free Document License (GFDL), versión 1.2 a Español (Castellano). No ha sido publicada por la Free Software Foundation y no establece legalmente los términos de distribución para trabajos que usen la GFDL (sólo el texto de la versión original en Inglés de la GFDL lo hace). Sin embargo, esperamos que esta traducción ayude los hispanohablantes a entender mejor la GFDL. La versión original de la GFDL está disponible en la Free Software Foundation (http://www.gnu.org/copyleft/fdl.html).

Esta traducción está basada en una de la versión 1.1 de Igor Támara y Pablo Reyes. Sin embargo la responsabilidad de su interpretación es de Joaquín Seoane.

Preámbulo

El propósito de esta Licencia es permitir que un manual, libro de texto, u otro documento escrito sea *libre* en el sentido de libertad: asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera comercial o no. En segundo término, esta Licencia proporciona al autor y al editor una manera de obtener reconocimiento por su trabajo, sin que se le considere responsable de las modificaciones realizadas por otros.

Esta Licencia es de tipo *copyleft*, lo que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Complementa la Licencia Pública General de GNU, que es una licencia tipo *copyleft* diseñada para el software libre.

Hemos diseñado esta Licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: un programa libre debe venir con manuales que ofrezcan las mismas libertades que el software. Pero esta licencia no se limita a manuales de software; puede usarse para cualquier texto, sin tener en cuenta su temática o si se publica como libro impreso o no. Recomendamos esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

1.0 APLICABILIDAD Y DEFINICIONES

Esta licencia se aplica a cualquier manual u otro trabajo, en cualquier soporte, que contenga una nota del propietario de los derechos de autor que indique que puede ser distribuido bajo los términos de esta licencia. Tal nota garantiza en cualquier lugar del mundo, sin pago de derechos y sin límite de tiempo, el uso de dicho trabajo según las condiciones aquí estipuladas.

En adelante la palabra Documento se referirá a cualquiera de dichos manuales o trabajos. Cualquier persona es un licenciatario y será referido como Usted. Usted acepta la licencia si copia, modifica o distribuye el trabajo de cualquier modo que requiera permiso según la ley de propiedad intelectual. Una Versión Modificada del Documento significa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones
y/o traducciones a otro idioma.

Una Sección Secundaria es un apéndice con título o una sección preliminar del Documento que trata exclusivamente de la relación entre los autores o editores y el tema general del Documento (o temas relacionados) pero que no contiene nada que entre directamente en dicho tema general (por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar nada de matemáticas). La relación puede ser una conexión histórica con el tema o temas relacionados, o una opinión legal, comercial, filosófica, ética o política acerca de ellos.

Las Secciones Invariantes son ciertas Secciones Secundarias cuyos títulos son designados como Secciones Invariantes en la nota que indica que el documento es liberado bajo esta Licencia. Si una sección no entra en la definición de Secundaria, no puede designarse como Invariante. El documento puede no tener Secciones Invariantes. Si el Documento no identifica las Secciones Invariantes, es que no las tiene.

Los Textos de Cubierta son ciertos pasajes cortos de texto que se listan como Textos de Cubierta Delantera o Textos de Cubierta Trasera en la nota que indica que el documento es liberado bajo esta Licencia. Un Texto de Cubierta Delantera puede tener como mucho 5 palabras, y uno de Cubierta Trasera puede tener hasta 25 palabras. Una copia Transparente del Documento, significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público en general, apto para que los contenidos puedan ser vistos y editados directamente con editores de texto genéricos o (para imágenes compuestas por puntos) con programas genéricos de manipulación de imágenes o (para dibujos) con algún editor de dibujos ampliamente disponible, y que sea adecuado como entrada para formateadores de texto o para su traducción automática a formatos adecuados para formateadores de texto.

Una copia hecha en un formato definido como Transparente, pero cuyo marcaje o ausencia de él haya sido diseñado para impedir o dificultar modificaciones posteriores por parte de los lectores no es Transparente. Un formato de imagen no es Transparente si se usa para una cantidad de texto sustancial. Una copia que no es Transparente se denomina Opaca.

Como ejemplos de formatos adecuados para copias Transparentes están ASCII puro sin marcaje, formato de entrada de Texinfo, formato de entrada de LaTeX, SGML o XML usando una DTD disponible públicamente, y HTML, PostScript o PDF simples, que sigan los estándares y diseñados para que los modifiquen personas.

Ejemplos de formatos de imagen transparentes son PNG, XCF y JPG. Los formatos Opacos incluyen formatos propietarios que pueden ser leídos y editados únicamente en procesadores de palabras propietarios, SGML o XML para los cuáles las DTD y/o

herramientas de procesamiento no estén ampliamente disponibles, y HTML, PostScript o PDF generados por algunos procesadores de palabras sólo como salida.

La Portada significa, en un libro impreso, la página de título, más las páginas siguientes que sean necesarias para mantener legiblemente el material que esta Licencia requiere en la portada. Para trabajos en formatos que no tienen página de portada como tal, Portada significa el texto cercano a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del texto. Una sección Titulada XYZ significa una parte del Documento cuyo título es precisamente XYZ o contiene XYZ entre paréntesis, a continuación de texto que traduce XYZ a otro idioma (aquí XYZ se refiere a nombres de sección específicos mencionados más abajo, como Agradecimientos, Dedicatorias, Aprobaciones o Historia. Conservar el Título de tal sección cuando se modifica el Documento significa que permanece una sección titulada XYZ según esta definición.

El Documento puede incluir Limitaciones de Garantía cercanas a la nota donde se declara que al Documento se le aplica esta Licencia. Se considera que estas Limitaciones de Garantía están incluidas, por referencia, en la Licencia, pero sólo en cuanto a limitaciones de garantía: cualquier otra implicación que estas Limitaciones de Garantía puedan tener es nula y no tiene efecto en el significado de esta Licencia.

2. Copia literal

Usted puede copiar y distribuir el Documento en cualquier soporte, sea en forma comercial o no, siempre y cuando esta Licencia, las notas de copyright y la nota que indica que esta Licencia se aplica al Documento se reproduzcan en todas las copias y que usted no añada ninguna otra condición a las expuestas en esta Licencia. Usted no puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

Usted también puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias públicamente.

3. Copiado en cantidad

Si publica copias impresas del Documento (o copias en soportes que tengan normalmente cubiertas impresas) que sobrepasen las 100, y la nota de licencia del Documento exige Textos de Cubierta, debe incluir las copias con cubiertas que lleven en forma clara y legible todos esos Textos de Cubierta: Textos de Cubierta Delantera en la cubierta delantera y Textos de Cubierta Trasera en la cubierta trasera. Ambas cubiertas deben identificarlo a Usted clara y legiblemente como editor de tales copias.

La cubierta debe mostrar el título completo con todas las palabras igualmente prominentes y visibles. Además puede añadir otro material en las cubiertas. Las copias con cambios limitados a las cubiertas, siempre que conserven el título del Documento y satisfagan estas condiciones, pueden considerarse como copias literales.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros (tantos como sea razonable colocar) en la verdadera cubierta y situar el resto en páginas adyacentes. Si Usted pública o distribuye copias Opacas del Documento cuya cantidad exceda las 100, debe incluir una copia Transparente, que pueda ser le da por una máquina, con cada copia

Opaca, o bien mostrar, en cada copia Opaca, una dirección de red donde cualquier usuario de la misma tenga acceso por medio de protocolos públicos y estandarizados a una copia Transparente del Documento completa, sin material adicional. Si usted hace uso de la última opción, deberá tomar las medidas necesarias, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio establecido por lo menos un año después de la última vez que distribuya una copia Opaca de esa edición al público (directamente o a través de sus agentes o distribuidores).

Se solicita, aunque no es requisito, que se ponga en contacto con los autores del Documento antes de redistribuir gran número de copias, para darles la oportunidad de que le proporcionen una versión actualizada del Documento.

4. Modificaciones

Puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las secciones 2 y 3 anteriores, siempre que usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada haciendo el rol del Documento, por lo tanto dando licencia de distribución y modificación de la Versión Modificada a quienquiera posea una copia de la misma.

Además, debe hacer lo siguiente en la Versión Modificada:

- Usar en la Portada (y en las cubiertas, si hay alguna) un título distinto al del Documento y de sus versiones anteriores (que deberían, si hay alguna, estar listadas en la sección de Historia del Documento). Puede usar el mismo título de versiones anteriores al original siempre y cuando quien las publicó originalmente otorgue permiso.

- Listar en la Portada, como autores, una o más personas o entidades responsables de la autoría de las modificaciones de la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (todos sus autores principales, si hay menos de cinco), a menos que le eximan de tal requisito.

- Mostrar en la Portada como editor el nombre del editor de la Versión Modificada.

- Conservar todas las notas de copyright del Documento.

- Añadir una nota de copyright apropiada a sus modificaciones, adyacente a las otras notas de copyright.

- Incluir, inmediatamente después de las notas de copyright, una nota de licencia dando el permiso para usar la Versión Modificada bajo los términos de esta Licencia, como se muestra en la Adenda al final de este documento.

- Conservar en esa nota de licencia el listado completo de las Secciones Invariantes y de los Textos de Cubierta que sean requeridos en la nota de Licencia del Documento original.

- Incluir una copia sin modificación de esta Licencia.

- Conservar la sección Titulada Historia, conservar su Título y añadirle un elemento que declare al menos el título, el año, los nuevos autores y el editor de la Versión Modificada, tal como figuran en la Portada. Si no hay una sección Titulada Historia en el Documento, crear una estableciendo el título, el año, los autores y el editor del Documento, tal como figuran en su Portada, añadiendo además un elemento describiendo la Versión Modificada, comose estableció en la oración anterior.

- Conservar la dirección en red, si la hay, dada en el Documento para el acceso público a una copia Transparente del mismo, así como las otras direcciones de red dadas en el Documento para versiones anteriores en las que estuviese basado. Pueden ubicarse en la sección Historia. Se puede omitir la ubicación en red de un trabajo que haya sido publicado por lo menos cuatro años antes que el Documento mismo, o si el editor original de dicha versión da permiso.

- En cualquier sección Titulada Agradecimientos o Dedicatorias, Conservar el Título de la sección y conservar en ella toda la sustancia y el tono de los agradecimientos y/o dedicatorias incluidas por cada contribuyente.

- Conservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Números de sección o el equivalente no son considerados parte de los títulos de la sección.
- Borrar cualquier sección titulada Aprobaciones. Tales secciones no pueden estar incluidas en las Versiones Modificadas.

- No cambiar el título de ninguna sección existente a Aprobaciones ni a uno que entre en conflicto con el de alguna Sección Invariante.

- Conservar todas las Limitaciones de Garantía.

Si la Versión Modificada incluye secciones o apéndices nuevos que califiquen como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, añada sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.

Puede añadir una sección titulada Aprobaciones, siempre que contenga únicamente aprobaciones de su Versión Modificada por otras fuentes –por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como la definición oficial de un estándar.

Puede añadir un pasaje de hasta cinco palabras como Texto de Cubierta Delantera y un pasaje de hasta 25 palabras como Texto de Cubierta Trasera en la Versión Modificada. Una entidad solo puede añadir (o hacer que se añada) un pasaje al Texto de Cubierta Delantera y uno al de Cubierta Trasera. Si el Documento ya incluye unos textos de cubiertas añadidos previamente por usted o por la misma entidad que usted representa, usted no puede añadir otro; pero puede reemplazar el anterior, con permiso explícito del editor que agregó el texto anterior.

Con esta Licencia ni los autores ni los editores del Documento dan permiso para usar sus nombres para publicidad ni para asegurar o implicar aprobación de cualquier Versión Modificada.

5. Combinación de documentos

Usted puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, listadas todas como Secciones Invariantes del trabajo combinado en su nota de licencia. Asimismo debe incluir la Limitación de Garantía.

El trabajo combinado necesita contener solamente una copia de esta Licencia, y puede reemplazar varias Secciones Invariantes idénticas por una sola copia. Si hay varias Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único añadiéndole al final del mismo, entre paréntesis, el nombre del autor o editor original de esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes de la nota de licencia del trabajo combinado.

En la combinación, debe combinar cualquier sección Titulada Historia de los documentos originales, formando una sección Titulada Historia; de la misma forma combine cualquier sección Titulada Agradecimientos, y cualquier sección Titulada Dedicatorias. Debe borrar todas las secciones tituladas Aprobaciones.

6. Colecciones de documentos

Puede hacer una colección que conste del Documento y de otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en todos los documentos por una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para cada copia literal de cada uno de los documentos en cualquiera de los demás aspectos.

Puede extraer un solo documento de una de tales colecciones y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraído, y siga esta Licencia en todos los demás aspectos relativos a la copia literal de dicho documento.

7. Agregación con trabajos independientes

Una recopilación que conste del Documento o sus derivados y de otros documentos o trabajos separados e independientes, en cualquier soporte de almacenamiento o distribución, se denomina un agregado si el copyright resultante de la compilación no se usa para limitar los derechos de los usuarios de la misma más allá de lo que los de los trabajos individuales permiten.

Cuando el Documento se incluye en un agregado, esta Licencia no se aplica a otros trabajos del agregado que no sean en sí mismos derivados del Documento.

Si el requisito de la sección 3 sobre el Texto de Cubierta es aplicable a estas copias del Documento y el Documento es menor que la mitad del agregado entero, los Textos de Cubierta del Documento pueden colocarse en cubiertas que enmarquen solamente el Documento dentro del agregado, o el equivalente electrónico de las cubiertas si el documento está en forma electrónica.

En caso contrario deben aparecer en cubiertas impresas enmarcando todo el agregado.

8. Traducción

La Traducción es considerada como un tipo de modificación, por lo que usted puede distribuir traducciones del Documento bajo los términos de la sección 4. El reemplazo de las Secciones Invariantes con traducciones requiere permiso especial de los dueños de derecho de autor, pero usted puede añadir traducciones de algunas o todas las Secciones Invariantes a las versiones originales de las mismas.

Puede incluir una traducción de esta Licencia, de todas las notas de licencia del documento, así como de las Limitaciones de Garantía, siempre que incluya también la versión en Inglés de esta Licencia y las versiones originales de las notas de licencia y Limitaciones de Garantía. En caso de desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la nota de licencia o la limitación de garantía, la versión original en Inglés prevalecerá.

Si una sección del Documento está Titulada Agradecimientos, Dedicatorias o Historia el requisito (sección 4) de Conservar su Título (Sección 1) requerirá, típicamente, cambiar su título.

9. Terminación

Usted no puede copiar, modificar, sublicenciar o distribuir el Documento salvo por lo permitido expresamente por esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y dará por terminados automáticamente sus derechos bajo esa Licencia. Sin embargo, los terceros que hayan recibido copias, o derechos, de usted bajo esta Licencia no verán terminadas sus licencias, siempre que permanezcan en total conformidad con ella.

10. Revisiones futuras de esta licencia

De vez en cuando la Free Software Foundation puede publicar versiones nuevas y revisadas de la Licencia de Documentación Libre GNU. Tales versiones nuevas serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar nuevos problemas o intereses. Vea http://www.gnu.org/copyleft/.

Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que se aplica una versión numerada en particular de esta licencia o cualquier versión posterior, usted tiene la opción de seguir los términos y condiciones de la versión especificada o cualquiera posterior que haya sido publicada (no como borrador) por la Free Software Foundation.

Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada (no como borrador) por la Free Software Foundation.

11.- ADENDA: Cómo usar esta Licencia en sus documentos

Para usar esta licencia en un documento que usted haya escrito, incluya una copia de la Licencia en el documento y ponga el siguiente copyright y nota de licencia justo después de la página de título:

> Copyright (c) AñO SU NOMBRE. Se concede permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre de GNU, Versión 1.2 o cualquier otra versión posterior publicada por la Free Software Foundation; sin Secciones Invariantes ni Textos de Cubierta Delantera ni Textos de Cubierta Trasera.
>
> Una copia de la licencia está incluida en la sección titulada GNU Free Documentation License.

Si tiene Secciones Invariantes, Textos de Cubierta Delantera y Textos de Cubierta Trasera, reemplace la frase sin... Trasera por esto:

> siendo las Secciones Invariantes LISTE SUS TÍTULOS, siendo los Textos de Cubierta Delantera LISTAR, y siendo sus Textos de Cubierta Trasera LISTAR.

Si tiene Secciones Invariantes sin Textos de Cubierta o cualquier otra combinación de los tres, mezcle ambas alternativas para adaptarse a la situación.

Si su documento contiene ejemplos de código de programa no triviales, recomendamos liberar estos ejemplos en paralelo bajo la licencia de software libre que usted elija, como la Licencia Pública General de GNU (GNU General Public License), para permitir su uso en software libre.