

# Apuntes de Django

Ezequiel Remus: < *ezequielremus@gmail.com* >

# Resumen

Este documento esta basado en la documentación de [Django](#) . En particular corresponde a la version Django 3.0, que admite Python 3.6 y versiones posteriores.

## Índice

<b>1. Introducción</b>	<b>3</b>
1.1. ¿Que es Django?	3
1.2. ¿Porqué usarlo?	3
<b>2. Iniciando un proyecto</b>	<b>3</b>
2.1. Entornos Virtuales	3
<b>3. Comenzando con Django</b>	<b>4</b>
3.1. Escribiendo tu primera aplicación Django , parte 1	4
3.1.1. Creando un Proyecto	4
3.1.2. El servidor de desarrollo	5
3.1.3. Creando la aplicación de encuestas (polls)	6
3.1.4. Escribiendo la primer vista (view)	7
3.2. Escribiendo tu primera aplicación Django , parte 2	9
3.2.1. Configuración de la Base de Datos	9
3.2.2. Creando un Modelo	10
3.2.3. Modelos de Activación	11
3.2.4. Jugando con la API	14
3.2.5. Introducción al administrador de Django	17
3.3. Escribiendo tu primer aplicación con Django , parte 3	20
3.3.1. Visión general	20
3.3.2. Escribiendo más vistas	21
3.3.3. Escribiendo vistas que realmente hacen algo	22
3.3.4. Errores 404	24
3.3.5. Usando el sistema de plantillas	25
3.3.6. Eliminando URLs codificadas en plantillas	26
3.3.7. Namespacing URL	26
3.4. Escribiendo tu primer aplicación en Django , parte 4	28
3.4.1. Escribiendo un pequeño formulario	28
3.4.2. Vistas Genéricas (Cuanto menos código, mejor!)	31
3.5. Escribiendo tu primera aplicación Django , parte 5	33
3.5.1. Testing: Introduciendo pruebas automatizadas	33
3.5.2. Estrategias básicas de Testing	34
3.5.3. Escribiendo nuestro primer Test	34
3.5.4. Corriendo Testeos	35
3.5.5. Arreglamos el Bug	36
3.5.6. Tests más completos	37
3.5.7. Testeando vistas	37
3.5.8. Algunas ideas para la creacion de otros tests	42
3.6. Escribiendo tu primera aplicación Django , parte 6	43
3.6.1. Personaliza el aspecto de tu aplicación	43
3.6.2. Imagen de fondo	44
<b>4. Referencias</b>	<b>45</b>
4.1. Refes de las partes de los tutoriales	45
4.2. Refes W3School	45
4.3. Atajos	45

# 1. Introducción

## 1.1. ¿Que es Django?

**Django** es un framework web diseñado para realizar aplicaciones de cualquier complejidad en unos tiempos muy razonables.

Está escrito en **Python** y tiene una comunidad muy amplia, que está en continuo crecimiento

## 1.2. ¿Porqué usarlo?

Los motivos principales para usar **Django** son:

- Es muy rápido : Si tenés una startup, estas apurado por terminar un proyecto o, simplemente, querés reducir costes, con **Django** *podéis construir una aplicación muy buena en poco tiempo.*
- Viene bien cargado : Cualquier cosa que necesitéis realizar, ya estará implementada, sólo hay que adaptarla a vuestras necesidades. Ya sea porque hay módulos de la comunidad, por cualquier paquete **Python** que encontréis o las propias aplicaciones que **Django** trae, que son muy útiles.
- Es bastante seguro : Podemos estar tranquilos con **Django** , ya que implementa por defecto algunas medidas de seguridad, las más clásicas, para que no haya **SQL Injection**, no haya *Cross site request forgery (CSRF)* o no haya **Clickjacking** por *JavaScript*. **Django** se encarga de manejar todo esto de una manera realmente sencilla.
- Es muy escalable : Podemos pasar desde muy poco a una aplicación enorme perfectamente, una aplicación que sea modular, que funcione rápido y sea estable.
- Es increíblemente versátil : Es cierto que en un principio **Django** comienza siendo un Framework para almacenar noticias por sitios de prensa, blogs y este estilo de webs, pero con el tiempo ha ganado tanta popularidad que se puede usar para el propósito que queráis.

*Otras bondades de **Django** que no se destacan en la web son:*

Su **ORM**, su interfaz para acceso a la base de datos , ya que hacer consultas con ella es una maravilla, es una herramienta muy buena.

Trae de serie un panel de administración, con el cual podemos dejar a personas sin ningún tipo de conocimiento técnico manejando datos importantes de una forma muy cómoda

# 2. Iniciando un proyecto

## 2.1. Entornos Virtuales

Lo primero y más importante es asegurarnos de crear un entorno para trabajar en nuestro proyecto.

Un entorno virtual es básicamente una abstracción la cual crea un conjunto vacío en **Python** , donde solo esta instalada la versión de **Python** que se utiliza junto con **pip** y las librerías básicas. En este conjunto podremos instalar todas las librerías que utilizaremos en el proyecto. Esto nos permitirá crear luego un archivo de referencia para conocer las librerías y versiones de estas utilizadas por el proyecto.

Existen varias formas de crear entornos virtuales:

### ■ Anaconda:

Es una distribución libre y abierta de los lenguajes **Python** y **R**, utilizada en ciencia de datos y aprendizaje automático (*Machine Learning*). Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos. Esta orientado a simplificar el despliegue y administración de los paquetes de software.

Las diferentes versiones de los paquetes se administran mediante el sistema de gestión de paquetes de **conda**, el cual lo hace bastante sencillo de instalar, correr y actualizar software de ciencia de datos y machine learning como ser *Scikit-team*, *Tensorflow* y *Scipy*.

La distribución Anaconda incluye más de 250 paquetes de ciencia de datos validos para **Windows**, **Linux** y **MacOs**.

**Referencias:** <https://docs.anaconda.com>

- **Virtualenv**: Es una herramienta para crear entornos de **Python** aislados, es decir entornos donde las librerías o las versiones de **Python** no interfieren con las carpetas que **Python** tiene por defecto en la máquina. Haciendo una analogía con un edificio, un entorno vendría siendo como una planta, usa ciertos recursos como el agua o la energía eléctrica (para el caso de **Python** usa la misma máquina) y a su vez cada planta tiene sus propios recursos, tales como los muebles, las habitaciones y demás (para el caso de **python** hablamos de librerías.)
- **pyenv**(Linux):

## 3. Comenzando con Django

### 3.1. Escribiendo tu primera aplicación Django , parte 1

A lo largo de este tutorial, nos guiaremos a través de la creación de una aplicación de encuesta básica. Consiste de dos partes:

- Un sitio público que permite a las personas ver encuestas y votar en ellas.
- Un sitio de administración que le permite agregar, cambiar y eliminar encuestas (*polls*).

Asumiremos que ya tiene **Django** instalado. Puede decir que **Django** está instalado y qué versión ejecuta el siguiente comando en un indicador de **shell** (indicado por el prefijo \$):

```
$ python -m django --version
```

Si **Django** está instalado, debería ver la versión de su instalación. Si no es así, recibirá un error que dice *"No module named django"* (Ningún módulo llamado django).

#### 3.1.1. Creando un Proyecto

Si es la primera vez que usa **Django** , tendrá que encargarse de la configuración inicial. Es decir, deberá generar automáticamente un código que establezca un proyecto de **Django** : una colección de configuraciones para una instancia de **Django** , incluida la configuración de la base de datos, las opciones específicas de **Django** y las configuraciones específicas de la aplicación.

Desde la línea de comando, Entre en un directorio donde le gustaría almacenar su código, luego ejecute el siguiente comando

```
$ django-admin startproject mysite
```

Esto creará un directorio **mysite** en su directorio actual. Si no funcionó, vea Problemas al ejecutar *django-admin*: <https://docs.djangoproject.com/en/3.0/faq/troubleshooting/#troubleshooting-django-admin>.

#### ¿Donde deberia estar este codigo?

Si su fondo está en **PHP** antiguo (sin el uso de marcos modernos), probablemente esté acostumbrado a poner código debajo de la raíz de documentos del servidor web (en un lugar como **/var/www**).

Con **Django** , no es necesario hacer eso. No es una buena idea colocar ninguno de estos códigos de **Python** en la raíz de documentos de su servidor web, ya que se corre el riesgo de que las personas puedan ver su código en la Web, lo cual no es bueno para la seguridad. Coloque su código en algún directorio fuera de la raíz del documento, como **/home/mycode**.

Al entrar a la carpeta **mysite** creada por el comando *startproject*, se crea el árbol de archivos de la figura de acá al costado

#### Nota:

Deberá evitar nombrar proyectos después de los componentes integrados de **Python** o Django. En particular, esto significa que debe evitar el uso de nombres como **django** (que entrará en conflicto con Django) o **test** (que entra en conflicto con un paquete Python incorporado).

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

Figura 1: Árbol de la carpeta **mysite**

Enumeremos estos archivos:

1. El directorio externo **mysite/root** es un contenedor para su proyecto. Su nombre no le importa a Django ; puedes cambiarle el nombre a lo que quieras
2. **manage.py**: es una utilidad de línea de comandos que te permite interactuar con este proyecto de Django de varias maneras. Puede leer todos los detalles sobre *manage.py* en [django-admin y manage.py](#).
3. El directorio **mysite/** interno es el paquete real de Python para su proyecto. Su nombre es el nombre del paquete Python que necesitará usar para importar cualquier cosa dentro de él (*por ejemplo, mysite.urls*).
4. **mysite/\_\_init\_\_.py**: un archivo vacío que le dice a Python que este directorio debe considerarse un paquete de Python. Si eres un principiante de Python , [lee mas sobre estos paquetes en la documentacion oficial Python](#)
5. **mysite/settings.py**: configuración para este proyecto de Django . La configuración de Django le dirá todo sobre cómo funciona la configuración ([django settings](#)).
6. **mysite/urls.py**: las declaraciones de URL para este proyecto de Django ; una "tabla de contenido" de su sitio impulsado por Django. Puede leer más sobre las URL en el [despachador de URL](#).
7. **mysite/asgi.py**: Un punto de entrada para servidores web compatibles con ASGI para servir su proyecto. Consulte [Cómo implementar con ASGI](#) para obtener más detalles.
8. **mysite/wsgi.py**: Un punto de entrada para servidores web compatibles con WSGI para servir su proyecto. Consulte [Cómo implementar con WSGI](#) para obtener más detalles.

### 3.1.2. El servidor de desarrollo

Verifiquemos que su proyecto Django funciona. Cambie al directorio externo de **mysite**, si aún no lo ha hecho, y ejecute el siguiente comando:

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

Luego, deberías ver sobre la línea de comandos, algo similar a lo que aparece en la siguiente imagen:

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are
applied.
Run 'python manage.py migrate' to apply them.

February 27, 2020 - 15:50:53
Django version 3.0, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figura 2: Inicio del servidor de django sobre la línea de comandos

Has iniciado el servidor de desarrollo Django , un servidor web ligero escrito exclusivamente en Python . Hemos incluido esto con Django para que pueda desarrollar las cosas rápidamente, sin tener que lidiar con la configuración de un servidor de producción, como Apache, hasta que esté listo para la producción.

Ahora es un buen momento para tener en cuenta que no debe usar este servidor en nada parecido a un entorno de producción. Está destinado solo para su uso durante el desarrollo.

Ahora que el servidor se está ejecutando, visite `http://127.0.0.1:8000/` con su navegador web. Verás un **¡Felicitaciones!** sobre la página y con un cohete despegando. Si es así entonces ¡Funcionó!.

## *Cambiando el puerto*

De manera predeterminada, el comando `runserver` inicia el servidor de desarrollo en la IP interna en el puerto 8000.

Si desea cambiar el puerto del servidor, páselo como un argumento de línea de comandos.

Por ejemplo, este comando inicia el servidor en el puerto 8080:

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

Si desea cambiar la IP del servidor, páselo junto con el puerto.

Por ejemplo, para escuchar todas las IP públicas disponibles

(lo cual es útil si está ejecutando Vagrant o desea mostrar su trabajo en otras computadoras en la red), use:

```
$ python manage.py runserver 0:8000
```

0 es un atajo para 0.0.0.0. Los documentos completos para el servidor de desarrollo se pueden encontrar en la referencia del servidor de ejecución.

## *Reinicio automático del servidor*

El servidor de desarrollo vuelve a cargar automáticamente el código [Python](#) para cada solicitud según sea necesario.

No necesita reiniciar el servidor para que los cambios de código surtan efecto.

Sin embargo, algunas acciones como agregar archivos no activan un reinicio, por lo que deberá reiniciar el servidor en estos casos.

### **3.1.3. Creando la aplicación de encuestas (polls)**

Una vez que su entorno de proyecto está listo y configurado, estamos listos para empezar a trabajar.

Cada aplicación que escribe en [Django](#) consiste en un paquete de [Python](#) que sigue una determinada convención. [Django](#) viene con una utilidad que genera automáticamente la estructura básica de directorios de una aplicación, por lo que puede centrarse en escribir código en lugar de crear directorios.

## *Proyectos Vs Aplicaciones*

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

Una aplicación es una aplicación web que hace algo, por ejemplo, un sistema de registro web, una base de datos de registros públicos o una pequeña aplicación de encuestas.

Un proyecto es una colección de configuraciones y aplicaciones para un sitio web en particular.

Un proyecto puede contener múltiples aplicaciones.

Una aplicación puede estar en múltiples proyectos.

Sus aplicaciones pueden vivir en cualquier lugar de su [ruta de Python](#). En este tutorial, crearemos nuestra aplicación de encuestas justo al lado de su archivo `manage.py` para que pueda importarse como su propio módulo de nivel superior, en lugar de un submódulo de `mysite`.

Para crear su aplicación, asegúrese de estar en el mismo directorio que `manage.py` y escriba este comando:

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

Eso creará un directorio de encuestas, que se presenta así:

```
polls/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
  models.py
  tests.py
  views.py
```

Figura 3: directorio de encuestas

Esta estructura de directorios albergará la aplicación de la encuesta.

#### 3.1.4. Escribiendo la primer vista (view)

Escribamos la primera vista. Abra el archivo **polls/views.py** y coloque el siguiente código de [Python](#) :

```
polls/views.py

from django.http import HttpResponseRedirect

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

Figura 4: Código [Python](#) de la primer vista

Esta es la vista más simple posible en [Django](#) . Para llamar a la vista, necesitamos asignarla a una **URL**, y para esto necesitamos una **URLconf**.

Para crear una **URLconf** en el directorio de encuestas, cree un archivo llamado *urls.py*. Su directorio de aplicaciones ahora debería verse como en la *figura 5* Donde en el archivo *urls.py* ubicado en *polls* se incluirá el código quedando este como el código de la *figura 5*.

<pre>polls/   __init__.py   admin.py   apps.py   migrations/     __init__.py   models.py   tests.py   urls.py   views.py</pre>	<pre>polls/urls.py  from django.urls import path  from . import views  urlpatterns = [     path('', views.index, name='index'), ]</pre>
--	---

Figura 5: Modificación del directorio polls y agregado del código sobre el *urls.py*

El siguiente paso es apuntar la **URLconf** raíz al módulo *polls.urls*. En **mysite/urls.py**, agregue una importación para [django.urls.include](#) e inserte un [include\(\)](#) en la lista *urlpatterns*. Debe quedar como en la *figura 6*.

```

from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]

```

Figura 6: Modificación del código sobre el `urls.py` del proyecto

La función `include()` permite hacer referencia a otros **URLconfs**. Cada vez que Django se encuentra con `include()`, corta cualquier parte de la **URL** que coincida con ese punto y envía la cadena restante a la **URLconf** incluida para su posterior procesamiento.

La idea detrás de `include()` es facilitar la conexión y reproducción de **URL**. Como las encuestas están en su propia **URLconf** (`polls/urls.py`), se pueden colocar debajo de `/polls/`, o debajo de `/fun_ polls/`, o debajo de `/content/polls/`, o cualquier otra ruta raíz, y la aplicación seguirá funcionando.

### *Cuando usamos include()*

Siempre debe usar `include()` cuando incluya otros patrones de **URL**. `admin.site.urls` es la única excepción a esto.

Ahora ha conectado una vista de índice en la **URLconf**. Verifique que esté funcionando con el siguiente comando:

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

Vaya a `http://localhost:8000/polls/` en su navegador, y debería ver el texto *"Hello, world. You're at the polls index"*, que definiste en la vista de índice.

### *Te aparece Page Not Found?*

Si recibe una página de error aquí, verifique que vaya a `http://localhost:8000/polls/` y no `http://localhost:8000/`.

La función `path()` recibe cuatro argumentos, dos obligatorios: ruta y vista, y dos opcionales: `kwargs` y nombre. En este punto, vale la pena revisar para qué sirven estos argumentos.

#### ■ *Argumento route del path():*

*route* es una cadena que contiene un patrón de **URL**. Al procesar una solicitud, Django comienza en el primer patrón en `urlpatterns` y avanza por la lista, comparando la **URL** solicitada con cada patrón hasta que encuentre uno que coincida.

Los patrones no buscan parámetros **GET** y **POST**, o el nombre de dominio. *Por ejemplo*, en una solicitud a `https://www.example.com/myapp/`, la **URLconf** buscará `myapp/`. En una solicitud a `https://www.example.com/myapp/?page=3`, la **URLconf** también buscará `myapp/`.

#### ■ *Argumento view del path():*

Cuando Django encuentra un patrón coincidente, llama a la función de vista especificada con un objeto `HttpRequest` como primer argumento y cualquier valor "capturado" de la ruta como argumentos de palabras clave. Vamos a dar un ejemplo de esto en un momento.



- **Argumento *kwargs* del *path()*:**

Los argumentos de palabras clave arbitrarias se pueden pasar en un diccionario a la vista de destino. No vamos a utilizar esta función de Django en el tutorial.

- **Argumento *name* del *path()*:**

Nombrar su **URL** le permite referirse a ella sin ambigüedades desde otros lugares de Django , especialmente desde las plantillas. Esta poderosa característica le permite realizar cambios globales en los patrones de **URL** de su proyecto mientras solo toca un solo archivo.

## 3.2. Escribiendo tu primera aplicación Django , parte 2

### 3.2.1. Configuración de la Base de Datos

Ahora, abra **mysite/settings.py**. Es un módulo Python normal con variables de nivel de módulo que representan la configuración de Django .

Por defecto, la configuración usa **SQLite**. Si eres nuevo en las bases de datos, o simplemente estás interesado en probar Django , esta es la opción más fácil. **SQLite** está incluido en Python , por lo que no necesitará instalar nada más para admitir su base de datos. Sin embargo, al comenzar su primer proyecto real, es posible que desee utilizar una base de datos más escalable como **PostgreSQL**, para evitar dolores de cabeza de cambio de base de datos en el futuro.

Si desea utilizar otra base de datos, instale los enlaces de base de datos apropiados ([database bindings](#)) y cambie las siguientes claves en el elemento **DATABASES** 'default' para que coincida con la configuración de conexión de su base de datos:

- **ENGINE** - Puede ser: `'django.db.backends.sqlite3'`, `'django.db.backends.postgresql'`, `'django.db.backends.mysql'`, or `'django.db.backends.oracle'`. Otros backends también están disponibles.
- **NAME** - El nombre de su base de datos. Si está utilizando **SQLite**, la base de datos será un archivo en su computadora; en ese caso, **NAME** debe ser la ruta absoluta completa, incluido el nombre de archivo, de ese archivo. El valor predeterminado, `os.path.join(BASE_DIR, 'db.sqlite3')`, almacenará el archivo en el directorio de su proyecto.

Si no está utilizando **SQLite** como su base de datos, se deben agregar configuraciones adicionales como **USUARIO** , **CONTRASEÑA** y **HOST** . Para obtener más detalles, consulte la documentación de referencia para **BASES DE DATOS**.

### *Para bases de datos que no sean SQLite*

Si está utilizando una base de datos además de **SQLite**, asegúrese de haber creado una base de datos en este momento. Haga eso con `"CREATE DATABASE database_name;"` dentro de la solicitud interactiva de su base de datos. También asegúrese de que el usuario de la base de datos proporcionado en **mysite/settings.py** tenga privilegios de crear base de datos". Esto permite la creación automática de una base de datos de prueba que será necesaria en un tutorial posterior.

Si está utilizando **SQLite**, no necesita crear nada de antemano; el archivo de la base de datos se creará automáticamente cuando sea necesario.

Mientras editas **mysite/settings.py**, configura **TIME\_ZONE** en su zona horaria.

Además, tenga en cuenta la configuración **INSTALLED\_APPS** en la parte superior del archivo. Esta contiene los nombres de todas las aplicaciones de Django que se activan en esta instancia de Django . Las aplicaciones se pueden usar en múltiples proyectos, y puede empaquetarlas y distribuirlas para que otras personas las usen en sus proyectos.

Por defecto, **INSTALLED\_APPS** contiene las siguientes aplicaciones, las cuales se instalan al instalar Django

- `django.contrib.admin` – Sitio de administración.
- `django.contrib.auth` – Sistema de autenticación.
- `django.contrib.contenttypes` – Un marco para los tipos de contenido.

- [django.contrib.sessions](#) – framework (marco) de sesiones.
- [django.contrib.messages](#) – framework (marco) de mensajes.
- [django.contrib.staticfiles](#) – Un marco para administrar archivos estáticos.

Estas aplicaciones se incluyen por defecto como una conveniencia para casos específicos.

Sin embargo, algunas de estas aplicaciones utilizan al menos una tabla de base de datos, por lo que debemos crear las tablas en la base de datos antes de poder usarlas. Para hacer eso, ejecuta el siguiente comando:

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

El comando migrate analiza la configuración [INSTALLED\\_APPS](#) y crea las tablas de base de datos necesarias de acuerdo con la configuración de la base de datos en su archivo **mysite/settings.py** y las migraciones de la base de datos que se envían con la aplicación (las cubriremos más adelante). Verá un mensaje para cada migración que aplique. Si está interesado, ejecute el cliente de línea de comandos para su base de datos y escriba \dt (PostgreSQL), MOSTRAR TABLAS; (MariaDB, MySQL), .schema (SQLite) o SELECT TABLE\_NAME FROM USER\_TABLES; (Oracle) para mostrar las tablas que Django creó.

### *Para los minimalistas*

Como dijimos anteriormente, las aplicaciones predeterminadas se incluyen para el caso común, pero no todos las necesitan. Si no necesita ninguno o todos ellos, no dude en comentar o eliminar las líneas apropiadas de [INSTALLED\\_APPS](#) antes de ejecutar la [migración](#).

El comando [migrate](#) solo ejecutará migraciones para aplicaciones en [INSTALLED\\_APPS](#).

#### 3.2.2. Creando un Modelo

### *Filosofía*

Un modelo es la fuente única y definitiva de verdad sobre sus datos. Contiene los campos y comportamientos esenciales de los datos que está almacenando.

[Django](#) sigue el principio DRY ([DRY principle](#)).

El objetivo es definir su modelo de datos en un lugar y derivar automáticamente cosas de él.

Esto incluye las migraciones, a diferencia de **Ruby On Rails**, por ejemplo, las migraciones se derivan completamente de su archivo de modelos, y son esencialmente un historial que [Django](#) puede recorrer para actualizar su esquema de base de datos para que coincida con sus modelos actuales.

En nuestra aplicación de encuestas, crearemos dos modelos: Pregunta (Question) y Elección (Choice).

```
polls/models.py

from django.db import models

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

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

Figura 7: Código de los modelos **Question** y **Choice**

Una `Question` tiene una pregunta y una fecha de publicación. Una opción tiene dos campos: el texto de la opción y un conteo de votos. Cada opción está asociada con una pregunta.

Estos conceptos están representados por las clases de `Python`. Edite el archivo `polls/models.py` para que se vea como en la *figura 7*.

Aquí, cada modelo está representado por una clase que subclasifica `django.db.models.Model`. Cada modelo tiene una serie de variables de clase, cada una de las cuales representa un campo de base de datos en el modelo.

Cada campo está representado por una instancia de clase `Campo`, por ejemplo, `CharField` para campos de caracteres y `DateTimeField` para fechas y horas. Esto le dice a `Django` qué tipo de datos contiene cada campo.

El nombre de cada instancia de campo (por ejemplo, `question_text` o `pub_date`) es el nombre del campo, en formato amigable para la máquina. Utilizará este valor en su código de `Python`, y su base de datos lo usará como el nombre de la columna.

Puede usar un primer argumento posicional opcional del campo para designar un nombre que sea legible por humanos. Eso se usa en un par de partes introspectivas de `Django`, y también sirve como documentación. Si no se proporciona este campo, `Django` usará el nombre legible por la máquina. En este ejemplo, solo hemos definido un nombre legible para humanos para `Question.pub_date`. Para todos los demás campos en este modelo, el nombre del campo legible por la máquina será suficiente como su nombre legible por humanos.

Algunas clases de campo tienen argumentos requeridos. `CharField`, por ejemplo, requiere que le des una **longitud máxima** (`max_length`). Eso se usa no solo en el esquema de la base de datos, sino también en la validación.

Un campo también puede tener varios argumentos opcionales; en este caso, hemos establecido el valor predeterminado (`default`) de votos en 0.

Finalmente, tenga en cuenta que se define una relación, usando `ForeignKey`. Eso le dice a `Django` que cada Elección está relacionada con una sola Pregunta. `Django` admite todas las relaciones de base de datos comunes: muchas a una, muchas a muchas y una a una.

### 3.2.3. Modelos de Activación

Ese pequeño fragmento de código del modelo le da a `Django` bastante información. Con él, `Django` es capaz de:

- Crear un esquema de base de datos (sentencias `CREATE TABLE`) para esta aplicación.
- Crear una **API** de acceso a la base de datos de `Python` para acceder a los objetos `Question` y `Choice`.

Pero primero debemos decirle a nuestro proyecto *que la aplicación de encuestas está instalada*.

## Filosofía

Las aplicaciones de `Django` son "conectables": puede usar una aplicación en varios proyectos y puede distribuir aplicaciones, ya que no tienen que estar vinculadas a una determinada instalación de `Django`.

Para incluir la aplicación en nuestro proyecto, necesitamos agregar una referencia a su clase de configuración en la configuración `INSTALLED_APPS`. La clase `PollsConfig` está en el archivo `polls/apps.py`, por lo que su ruta punteada es `'polls.apps.PollsConfig'`. Hay que editar el archivo `mysite/settings.py` y agregar esa ruta punteada a la configuración `INSTALLED_APPS`. Se verá así:

```
mysite/settings.py

INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Figura 8: Instalación de la aplicación `polls` en el `settings.py` del proyecto

Listo, **Django** con esto ya sabe incluir la aplicación de encuestas. Ahora, debemos aplicar migraciones, lo cual se hace con el comando:

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

Luego, en la terminal vas a ver algo similar a:

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

Figura 9: Terminal con un makemigrations satisfactorio

Al correr *makemigrations*, le estás diciendo a Django que has realizado algunos cambios en tus modelos (En este caso, agregaste modelos al **models.py**) y que queremos que esos cambios sean almacenados como migraciones.

Las migraciones es la forma en la cual **Django** almacena los cambios en sus modelos (y, por lo tanto, en el esquema de su base de datos): estos son archivos en el disco. Podemos leer las migraciones si queremos; en este caso son las encuestas de **polls/migrations/0001\_initial.py**. No se preocupe, no se espera que los lea cada vez que Django hace uno, pero están diseñados para ser editados por humanos en caso de que desee modificar manualmente cómo **Django** cambia las cosas.

Hay un comando que ejecutará las migraciones por usted y administrará el esquema de su base de datos automáticamente, este es el comando *migrate*, y lo veremos en un momento, pero primero, veamos qué **SQL** ejecutará esa migración. El comando *sqlmigrate* toma nombres de migración y devuelve su SQL:

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

Vas a ver algo similar a esto, quizás en otro formato pero con el mismo contenido:

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

Figura 10: python manage.py sqlmigrate polls 0001

Notemos que:

- El resultado exacto variará según la base de datos que esté utilizando. El ejemplo anterior se genera para PostgreSQL.
- Los nombres de las tablas se generan automáticamente combinando el nombre de la aplicación (encuestas) y el nombre en minúsculas del modelo: pregunta y elección. (Puede anular este comportamiento).
- Las claves primarias (ID) se agregan automáticamente. (También puede anular esto).
- Por convención, Django agrega `_id` al nombre del campo de clave externa. (Sí, también puedes sobrescribirlo).
- La relación de clave externa se hace explícita por una restricción FOREIGN KEY. No se preocupe por las partes DEFERRABLES; esto le dice a PostgreSQL que no aplique la clave foránea hasta el final de la transacción
- Se adapta a la base de datos que está utilizando, por lo que los tipos de campo específicos de la base de datos como `auto_increment` (MySQL), `serial` (PostgreSQL) o `autoincrement` de clave primaria entera (SQLite) se manejan automáticamente. Lo mismo ocurre con las citas de los nombres de campo, por ejemplo, con comillas dobles o comillas simples.
- El comando `sqlmigrate` no ejecuta realmente la migración en su base de datos, sino que lo imprime en la pantalla para que pueda ver lo que **SQL Django** cree que es necesario. Es útil para verificar qué hará Django o si tiene administradores de bases de datos que requieren *scripts SQL* para dichos cambios.

Si está interesado, también puede ejecutar `python manage.py check`; Esto verifica si hay algún problema en su proyecto sin hacer migraciones o tocar la base de datos.

Ahora, debemos correr el comando `migrate` otra vez para crear estos modelos en las tablas de tu base de datos.

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

Figura 11: `python manage.py sqlmigrate polls 0001`

El comando `migrate` toma todas las migraciones que no se han aplicado (Django rastrea cuáles se aplican usando una tabla especial en su base de datos llamada `django_migrations`) y las ejecuta en su base de datos, esencialmente, sincronizando los cambios que realizó en sus modelos con el esquema en la base de datos.

Las migraciones son muy potentes y le permiten cambiar sus modelos con el tiempo, a medida que desarrolla su proyecto, sin la necesidad de eliminar su base de datos o tablas y crear nuevas: se especializa en actualizar su base de datos en vivo, sin perder datos. Los cubriremos con más profundidad en una parte posterior del tutorial, pero por ahora, recuerde la guía de tres pasos para realizar cambios en el modelo:

- Creamos/modificamos los modelos en el archivo `models.py`.
- Luego corremos el comando `python manage.py makemigrations` para crear las migraciones para esos cambios.
- Corremos el comando `python manage.py migrate` para aplicar los cambios realizados en la base de datos.

La razón por la que existen comandos separados para realizar y aplicar migraciones es porque comprometerás las migraciones en tu sistema de control de versiones y las enviarás con tu aplicación; no solo facilitan su desarrollo, también pueden ser utilizados por otros desarrolladores y en producción.

### 3.2.4. Jugando con la API

Ahora, entremos al *shell* interactivo de [Python](#) y juguemos con la **API** gratuita que [Django](#) le brinda. Para invocar el *shell* de [Python](#), use este comando:

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

Estamos usando esto en lugar de simplemente escribir "*python*", porque *manage.py* establece la variable de entorno `DJANGO_SETTINGS_MODULE`, que le da a [Django](#) la ruta de importación de [Python](#) a su archivo `mysite/settings.py`.

Una vez que entramos en la *shell* podemos trabajar con ella realizando [consultas](#) como se indica en la figura:

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

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

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

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

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

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

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

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>
```

Getting Help

Language: en

Documentation version: 3.0

Figura 12: Making queries

Ahora, vemos una cosa `<Question: Question object (1)>` no es una representación útil de este objeto. Arreglemos eso editando el modelo `Question` (en el archivo `polls/models.py`) y agregando un método `__str__()` a los modelos `Question` y `Choice`:

```
polls/models.py

from django.db import models

class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text

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

Figura 13: ●

Es importante agregar los métodos `__str__()` a sus modelos, no solo para su propia conveniencia al tratar con el mensaje interactivo, sino también porque las representaciones de los objetos se utilizan en todo el administrador generado automáticamente por [Django](#).

Agreguemos también un método personalizado a este modelo:

```
polls/models.py

import datetime

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

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

Figura 14: ●

Observe la adición de `import datetime` y de `django.utils import timezone`, para hacer referencia al módulo de fecha y hora estándar de [Python](#) y las utilidades relacionadas con la zona horaria de [Django](#) en `django.utils.timezone`, respectivamente. Si no está familiarizado con el manejo de zona horaria en [Python](#), puede obtener más información en los documentos de soporte de zona horaria.

Guarde estos cambios e inicie un nuevo *shell* interactivo de [Python](#) ejecutando nuevamente `python manage.py shell`:

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

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

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



```

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

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

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

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

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

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

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

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

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

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

# Let's delete one of the choices. Use delete() for that.
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()

```



Para obtener más información sobre las relaciones del modelo, consulte Acceso a objetos relacionados ([Accessing related objects](#)). Para obtener más información sobre cómo utilizar guiones bajos dobles para realizar búsquedas de campo a través de la API, consulte las búsquedas de campo ([field lookups](#)). Para obtener detalles completos sobre la API de base de datos, consulte nuestra referencia de API de base de datos ([Database API reference](#)).

### 3.2.5. Introducción al administrador de Django

#### *Filosofía*

Generar sitios de administración para que su personal o clientes agreguen, cambien y eliminen contenido es un trabajo tedioso que no requiere mucha creatividad. Por esa razón, Django automatiza por completo la creación de interfaces de administración para modelos.

Django fue escrito en un entorno de redacción, con una separación muy clara entre los “editores de contenido” y el sitio “público”.

Los administradores del sitio usan el sistema para agregar noticias, eventos, resultados deportivos, etc., y ese contenido se muestra en el sitio público.

Django resuelve el problema de crear una interfaz unificada para que los administradores del sitio editen contenido.

El administrador no está destinado a ser utilizado por los visitantes del sitio.

Es para los administradores del sitio.

#### Creando un usuario administrador

Crear un usuario administrador Primero, necesitaremos crear un usuario que pueda iniciar sesión en el sitio de administración. Ejecute el siguiente comando:

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

Al correr este comando nos pedirá una serie de datos, como el nombre del *usuario*, *nuestro mail* y una *contraseña*.

#### Corremos el servidor de desarrollo

El administrador de Django ya viene activado por defecto. Primero debemos correr el servidor una vez creado el administrador:

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

Ahora, abra un navegador web y vaya a / **admin** / en su dominio local, por ejemplo, **http://127.0.0.1:8000/admin/**. Debería ver la pantalla de inicio de sesión del administrador:

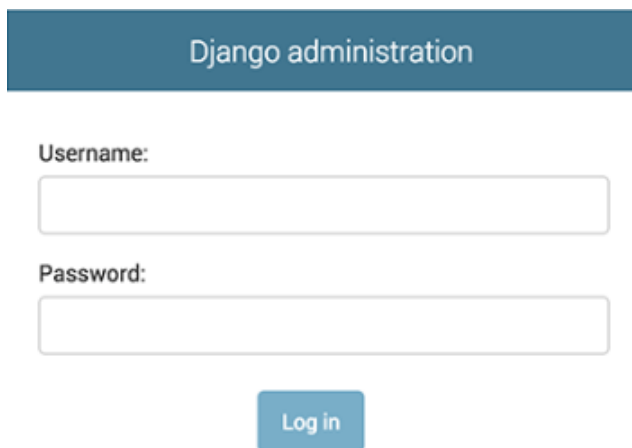


Figura 15: Login del administrador

Dado que la traducción ([translation](#)) está activada de manera predeterminada, la pantalla de inicio de sesión puede mostrarse en su propio idioma, según la configuración de su navegador y si [Django](#) tiene una traducción para este idioma.

## El sitio de administración

Ahora, intente iniciar sesión con la cuenta de *superusuario* que creó en el paso anterior. Debería ver la página de índice de administración de [Django](#):

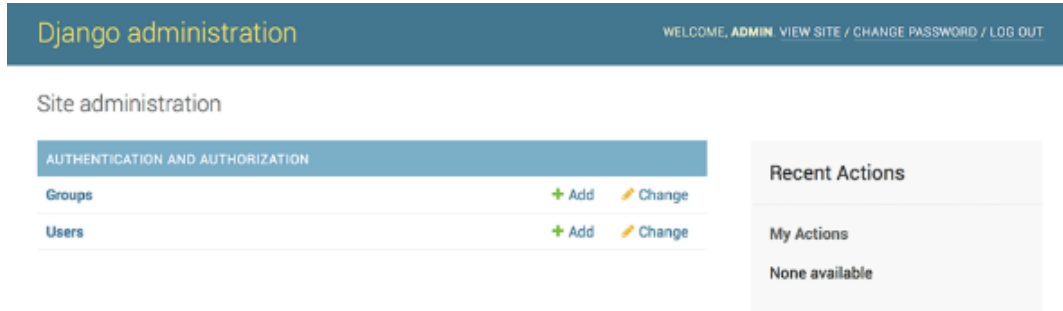


Figura 16: Login del administrador

Debería ver algunos tipos de contenido editable: grupos y usuarios. Estos los proporciona [django.contrib.auth](#), el cual es el *marco de autenticación* enviado por [Django](#).

## Hacer que la aplicación de la aplicacion sea modificable en el administrador

¿Pero dónde está nuestra aplicación de encuestas? No se muestra en la página de índice de administrador.

Solo una cosa más que hacer: necesitamos decirle al administrador que los objetos de Pregunta tienen una interfaz de administrador. Para hacer esto, abra el archivo `polls/admin.py` y edítelo para que se vea así:

```
polls/admin.py

from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

Figura 17: Login del administrador

## Explore la funcionalidad de administración gratuita

Ahora que hemos registrado el modelo **Question**, [Django](#) sabe que debería mostrarse en la página de índice de administración:



Figura 18: Login del administrador

Haga clic en “**Question**”. Ahora está en la página de “lista de cambios” para Question. Esta página muestra todas las preguntas en la base de datos y le permite elegir una para cambiarla. Ahí está el “¿Qué pasa?” pregunta que creamos anteriormente:

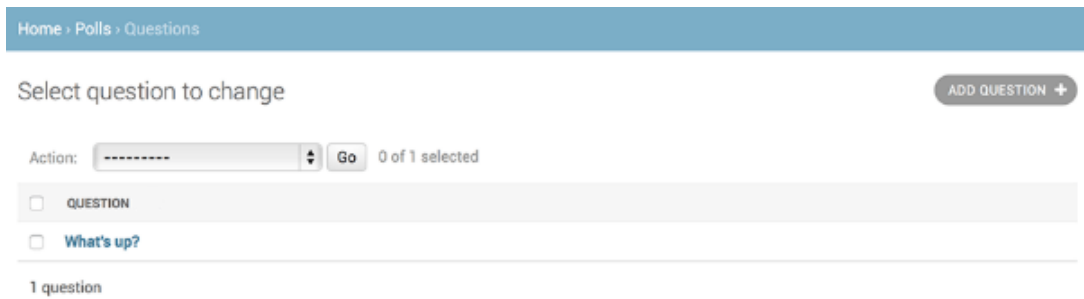


Figura 19: Login del administrador

Haz clic en “¿Qué pasa?” pregunta para editarlo:

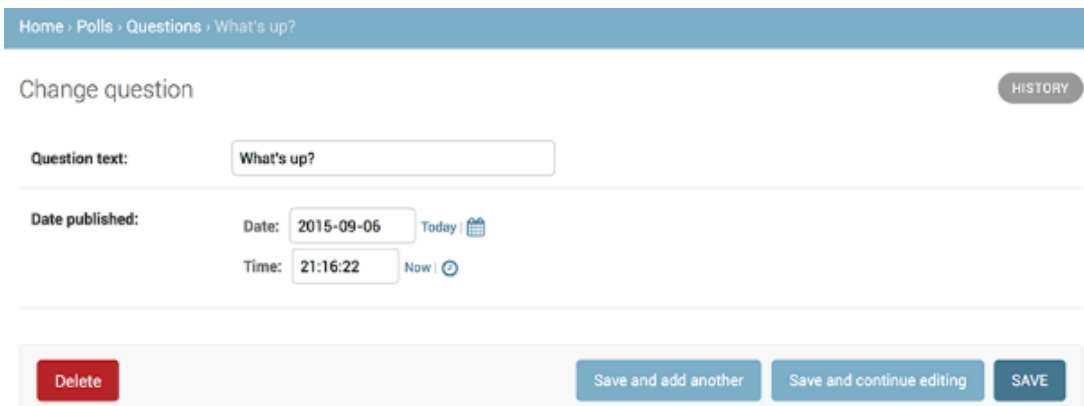


Figura 20: Modificando la Question

Acá debemos tener en cuenta que:

- El formulario se genera automáticamente a partir del modelo de Pregunta.
- Los diferentes tipos de campo de modelo (`DateTimeField`, `CharField`) corresponden al *widget* de entrada **HTML** apropiado. Cada tipo de campo sabe cómo mostrarse en el administrador de Django.
- Cada `DateTimeField` obtiene accesos directos de **JavaScript** gratuitos. Las fechas obtienen un acceso directo “Hoy” una ventana emergente de calendario, y las horas obtienen un acceso directo “Ahora” una ventana emergente conveniente que enumera las horas comúnmente ingresadas.

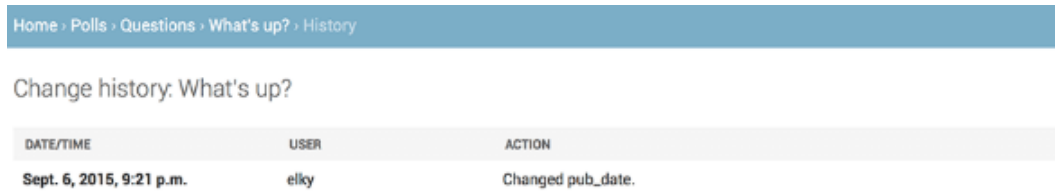
La parte inferior de la página le ofrece un par de opciones:

- 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 y vuelve a cargar la página de administración para este objeto.
- Guardar y agregar otro: guarda los cambios y carga un nuevo formulario en blanco para este tipo de objeto
- Eliminar: muestra una página de confirmación de eliminación

Si el valor de “Fecha de publicación” no coincide con la hora en que creó la pregunta en el Tutorial 1, probablemente significa que olvidó establecer el valor correcto para la configuración `TIME_ZONE`. Cámbielo, vuelva a cargar la página y verifique que aparezca el valor correcto.

Cambie la “Fecha de publicación” haciendo clic en los accesos directos “Hoy” “Ahora”. Luego haga clic en “Guardar y continuar editando”. Luego haga clic en “Historial” en la esquina superior derecha. Verá una página que enumera

todos los cambios realizados en este objeto a través del administrador de Django, con la marca de tiempo y el nombre de usuario de la persona que realizó el cambio:



DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

Figura 21: Modificando la Question

Cuando se sienta cómodo con la API de modelos y se haya familiarizado con el sitio de administración, lea la [parte 3](#) de este tutorial para obtener información sobre cómo agregar más vistas a nuestra aplicación de encuestas.

### 3.3. Escribiendo tu primer aplicación con Django , parte 3

#### 3.3.1. Visión general

Una vista es un “tipo” de página web en su aplicación Django que generalmente cumple una función específica y tiene una plantilla específica. *Por ejemplo*, en una aplicación de **blog**, puede tener las siguientes vistas:

- **HomePage del Blog** : Muestra las últimas entradas.
- **Página de “detail” de entrada** : página de enlace permanente para una sola entrada.
- **Página de archivo basada en el año** : muestra todos los meses con entradas en el año dado.
- **Página de archivo basada en el mes** : muestra todos los días con entradas en el mes dado.
- **Página de archivo basada en el día** : muestra todas las entradas en el día dado.
- **Acción de comentario** : maneja la publicación de comentarios en una entrada determinada.

En nuestra aplicación de encuesta, tendremos las siguientes cuatro vistas:

- **Página de “índice” (index ) de preguntas**: muestra las últimas preguntas.
- **Página de “detalles” (detail) de la pregunta** : muestra un texto de pregunta, sin resultados pero con un formulario para votar.
- **Página de “resultados” (result) de la pregunta** : Muestra un resultado para cada Question en particular.
- **Accion de voto** : maneja la votación de una elección particular en una pregunta particular.

En Django , las páginas web y otros contenidos se entregan mediante vistas. Cada vista está representada por una función de Python (o método, en el caso de vistas basadas en clases). Django elegirá una vista examinando la URL solicitada (para ser precisos, la parte de la URL después del nombre de dominio).

Ahora, en su tiempo en la web, puede haber encontrado cositas lindas como `ME2/Sites/dirmod.asp?Sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B`. Te complacerá saber que Django nos permite patrones de URL mucho más elegantes que eso.

Un patrón de URL es la forma general de una URL, por ejemplo: `/newsarchive/<año>/<mes>/`.

Para pasar de una URL a una vista, Django utiliza lo que se conoce como “URLconfs”. Un URLconf asigna patrones de URL a vistas.

Este tutorial proporciona instrucciones básicas sobre el uso de URLconfs, y puede consultar en [URL info](#) para obtener más información.

### 3.3.2. Escribiendo más vistas

Ahora, agreguemos unas vistas más a `polls/viwes.py`. Estas vistas serán ligeramente diferentes, pues estas tomarán el `id` de una **Question** como argumento además de el parámetro obligatorio `request`:

```
polls/views.py

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

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

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

Figura 22: Agregamos vistas

Instala estas nuevas vistas en el módulo `polls/urls` agregando los siguientes `path()`

```
polls/urls.py

from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: /polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: /polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Figura 23: Agregamos vistas

Eche un vistazo en su navegador, en `"/polls/34/"`. Ejecutará el método `detail()` y mostrará cualquier ID que proporciones en la URL. Pruebe `"/polls/34/results/"` y `"/polls/34/vote/"` también; estos mostrarán los resultados de marcador de posición y las páginas de votación.

Cuando alguien solicita una página de su sitio web, por ejemplo, `"/polls/34/"`, Django cargará el módulo de `Python mysite.urls` porque está indicado por la configuración `ROOT_URLCONF`. Encuentra la variable llamada `urlpatterns` y atraviesa los patrones en orden. Después de encontrar la coincidencia en `'polls/'`, elimina el texto coincidente (`"polls/"`) y envía el texto restante `- "34/"` a la `URLconf "polls.urls"` para su posterior procesamiento. Allí coincide con `'<int: question_id>/'`, lo que resulta en una llamada a la vista de `detail()` así:

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

La parte `question_id = 34` proviene de `<int: question_id>`. El uso de corchetes angulares “captura” parte de la URL y la envía como argumento de palabra clave a la función de vista. La parte: `question_id>` de la cadena define el nombre que se utilizará para identificar el patrón coincidente, y la `<int: part` es un convertidor que determina qué patrones deben coincidir con esta parte de la ruta URL.

No es necesario agregar **URL cruft** como **.html**, a menos que lo desee, en cuyo caso puede hacer algo como esto:

```
path('polls/latest.html', views.index),
```

Pero no hagas eso, no es ni necesario ni comodo a la vista.

### 3.3.3. Escribiendo vistas que realmente hacen algo

Cada vista es responsable de hacer una de dos cosas: devolver un objeto [HttpResponse](#) que contenga el contenido de la página solicitada, o generar una excepción como [Http404](#) . El resto depende de usted

Su vista puede leer registros de una base de datos, o no. Puede usar un sistema de plantillas como el de [Django](#) , o un sistema de plantillas [Python](#) de terceros, o no. Puede generar un archivo **PDF**, generar **XML**, crear un archivo **ZIP** sobre la marcha, lo que desee, utilizando las bibliotecas de [Python](#) que desee.

Todo lo que [Django](#) quiere es que [HttpResponse](#). O una excepción.

Debido a que es conveniente, usemos la propia **API** de base de datos de [Django](#) , que cubrimos en la *parte 2*. Aquí hay una puñalada en una nueva vista de *índice()*, que muestra las últimas 5 preguntas de encuesta en el sistema, separadas por comas, según la fecha de publicación

```
polls/views.py

from django.http import HttpResponse

from .models import Question

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

# Leave the rest of the views (detail, results, vote) unchanged
```

Figura 24: Modificamos la función *index*

Sin embargo, hay un problema aquí: el diseño de la página está codificado en la vista. Si desea cambiar el aspecto de la página, deberá editar este código de [Python](#) . Entonces, usemos el sistema de plantillas de [Django](#) para separar el diseño de [Python](#) creando una plantilla que la vista pueda usar.

Primero, cree un directorio llamado *plantillas* en su directorio de encuestas. [Django](#) buscará plantillas allí.

La configuración de **TEMPLATES** (plantillas) de su proyecto describe cómo [Django](#) cargará y renderizará las plantillas. El archivo de configuración predeterminado configura un *back-end* de [DjangoTemplates](#) cuya opción **APP\_DIRS** está establecida en **True**. Por convención, [DjangoTemplates](#) busca un subdirectorio de "plantillas".<sup>en</sup> cada una de las **INSTALLED\_APPS**.

Dentro del directorio de plantillas que acaba de crear, cree otro directorio llamado *encuestas*, y dentro de ese, cree un archivo llamado *index.html*. En otras palabras, su plantilla debe estar en **polls/templates/polls/index.html**. Debido a cómo funciona el cargador de plantillas **app\_directories** como se describe anteriormente, puede referirse a esta plantilla dentro de [Django](#) como **polls/index.html**.

### *Espacio de nombres de plantillas (Template namespacing)*

Ahora podríamos evitar poner nuestras plantillas directamente en **polls/templates** (en lugar de crear otro subdirectorio de encuestas), pero en realidad sería una mala idea.

[Django](#) elegirá la primera plantilla que encuentre cuyo nombre coincida, y si tuviera una plantilla con el mismo nombre en una aplicación diferente, [Django](#) no podría distinguir entre ellas. Necesitamos poder apuntar a [Django](#) al correcto, y la mejor manera de asegurar esto es espaciando los nombres. Es decir, colocando esas plantillas dentro de otro directorio nombrado para la aplicación misma.

Pon el siguiente código en esa plantilla:

```
polls/templates/polls/index.html

{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Figura 25: Modificamos la función *index*

## Nota

Para acortar el tutorial, todos los ejemplos de plantillas usan HTML incompleto.  
En sus propios proyectos, debe usar [documentos HTML completos](#).

Ahora actualicemos nuestro *index* en **polls/views.py** para que use el la platnilla *index.html*

```
polls/views.py

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

from .models import Question

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

Figura 26: Agregamos el *loader* para cargar el *template* en la función *index*

Ese código carga la plantilla llamada *polls/index.html* y le pasa un contexto. El contexto es una plantilla de mapeo de diccionario de nombres de variables a objetos de [Python](#) .

Cargue la página apuntando su navegador a **"/polls/"**, y debería ver una lista con viñetas que contiene la pregunta "¿Qué pasa?".<sup>en la parte 2</sup>. El enlace apunta a la página de detalles de la pregunta.

### Otra forma es renderisar ([shourcat render\(\)](#))

Existe una forma muy común cargar una plantilla, llenar un contexto y devolver un objeto [HttpResponse](#) con el resultado de la plantilla renderizada. Django proporciona un acceso directo. Aquí está la vista completa del índice (), reescrita:

```
polls/views.py

from django.shortcuts import render

from .models import Question

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

Figura 27: Agregamos el *loader* para cargar el *template* en la función *index*

Tenga en cuenta que una vez que hayamos hecho esto en todas estas vistas, ya no necesitamos importar el cargador (*loader*) y *HttpResponse* (querrá mantener *HttpResponse* si todavía tiene los métodos de código auxiliar para obtener detalles, resultados y votar).

La función *render()* toma el objeto de solicitud como su primer argumento, un nombre de plantilla como su segundo argumento y un diccionario como su tercer argumento opcional. Devuelve un objeto *HttpResponse* de la plantilla dada representada con el contexto dado.

### 3.3.4. Errores 404

Ahora, abordemos la vista detallada de la pregunta: la página que muestra el texto de la pregunta para una encuesta determinada. Aquí está la vista:

```
polls/views.py

from django.http import Http404
from django.shortcuts import render

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

Figura 28: Levantamiento de un Error 404 sobre el view

El nuevo concepto aquí: la vista genera (*raise*) la excepción *Http404* si no existe una pregunta con el **ID** solicitado.

Discutiremos un poco más adelante qué podría poner en esa plantilla de **polls/detail.html**, pero si desea que el ejemplo anterior funcione rápidamente, un archivo que contiene solo:

```
polls/templates/polls/detail.html

{{ question }}
```

Figura 29: template del *detail.html*



### Shortcut: `get_object_or_404()`

Es muy común utilizar un `get()` para generar un `Http404` si es que el objeto no existe en la base de datos. Django provee un *shortcut* para este tipo de cosas, este es el `get_object_or_404()`. El código utilizando esta herramienta nos quedaría como sigue:

```
polls/views.py

from django.shortcuts import get_object_or_404, render

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

Figura 30: Función detail

La función `get_object_or_404()` toma un *Django model* como primer argumento y un número arbitrario de argumentos de palabras clave, que pasa a la función `get()` del administrador del modelo. Si el modelo no existe genera un `Http404`.

## Filosofía

¿Por qué utilizamos una función auxiliar `get_object_or_404()` en lugar de detectar automáticamente las excepciones `ObjectDoesNotExist` en un nivel superior, o hacer que la API del modelo genere `Http404` en lugar de `ObjectDoesNotExist`?  
Porque eso acoplaría la capa del modelo a la capa de vista.  
Uno de los principales objetivos de diseño de Django es mantener un acoplamiento flexible.  
Se introduce algún acoplamiento controlado en el módulo `django.shortcuts`.

También hay una función `get_list_or_404()`, que funciona igual que `get_object_or_404()`, excepto que usa `filter()` en lugar de `get()`. Eleva `Http404` si la lista está vacía.

### 3.3.5. Usando el sistema de plantillas

Volviendo a la vista del *detail* para nuestra aplicación de encuestas. Dada la pregunta de la variable de contexto, así es como se vería la plantilla `polls/detail.html`:

```
polls/templates/polls/detail.html

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

Figura 31: template del *detail.html*

El sistema de plantillas utiliza la sintaxis de búsqueda de puntos para acceder a los atributos variables. En el ejemplo de `{{ question.question_text }}`, primero Django realiza una búsqueda de diccionario en la pregunta del objeto. De lo contrario, intenta una búsqueda de atributos, que en este caso, funciona. Si la búsqueda de atributos hubiera fallado, habría intentado una búsqueda de índice de lista.

La llamada al metodo ocurre en el bucle `{{ % for % }}`: `question.choice_set.all` se interpreta como el código Python `question.choice_set.all()` , que devuelve un iterable de objetos **Choice** y se itera con un `{{ % for % }}`.

En la [Guia de templates](#) vas a encontrar más info.

### 3.3.6. Eliminando URLs codificadas en plantillas

Recordemos, cuando escribimos el link hacia el template `polls/index.html`, el link estaba codificado (hardcodeado) masomenos así:

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

El problema con esta codificación aproximada es que se convierte en un desafío cambiar las **URL** en proyectos con muchas plantillas. Sin embargo, dado que definió el argumento de nombre en las funciones `path()` en el módulo `polls.urls`, puede eliminar la dependencia de rutas **URL** específicas definidas en sus configuraciones de **URL** utilizando la etiqueta de plantilla `{ % url % }`:

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

La forma en que esto funciona es buscando la definición de **URL** como se especifica en el módulo `polls.urls`. Puede ver exactamente dónde se define a continuación el nombre de **URL** de "detail"

```
...
# the 'name' value as called by the {% url %} template tag
path('<int:question_id>/', views.detail, name='detail'),
...
```

Si desea cambiar la **URL** de la vista de detalles de las encuestas a otra, tal vez a algo como `polls/detail/12/` en lugar de hacerlo en la plantilla (o plantillas), debe cambiarla en `polls/urls.py`:

```
...
# added the word 'specifics'
path('specifics/<int:question_id>/', views.detail, name='detail'),
...
```

### 3.3.7. Namespacing URL

El proyecto tiene solo una aplicación, `polls`. En proyectos reales de **Django** , puede haber cinco, diez, veinte aplicaciones o más. ¿Cómo diferencia Django los nombres de URL entre ellos? Por ejemplo, la aplicación de encuestas tiene una vista detallada, y también podría tener una aplicación en el mismo proyecto que es para un **blog**. ¿Cómo se logra que **Django** sepa qué vista de aplicación crear para una **URL** cuando se usa la etiqueta de plantilla `{ % url % }` ?

La respuesta es *agregar namespaces* al **URLconf** en `polls/urls.py`. Para esto vamos a `polls/urls.py` y agregamos la variable `app_name` para setear el namespace de la aplicación:

```
polls/urls.py

from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Figura 32: Agregamos el namespace a **polls/urls.py**

Luego, modificamos el template del index de la siguiente forma:

```
polls/templates/polls/index.html

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

Figura 33: Agregamos el namespace a **polls/urls.py**

y para apuntar al namespace de la vista del detail:

```
polls/templates/polls/index.html

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

Figura 34: Agregamos el namespace a **polls/urls.py**

### 3.4. Escribiendo tu primer aplicación en Django , parte 4

#### 3.4.1. Escribiendo un pequeño formulario

Actualicemos nuestra plantilla de detalles de la encuesta (`polls/detail.html`) del último tutorial, de modo que la plantilla contenga un elemento **HTML** `<form>`:

```
polls/templates/polls/detail.html

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

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

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

Figura 35: Agregamos el namespace a `polls/urls.py`

Resumiendo:

- La plantilla anterior muestra un botón de opción para cada elección de pregunta. El valor de cada botón de opción es la ID de la opción de pregunta asociada. El nombre de cada botón de radio es **Choice**. Eso significa que, cuando alguien selecciona uno de los botones de opción y envía el formulario, enviará la opción **POST** de **Choice=#** donde # es el ID de la opción seleccionada. Este es el concepto básico de los formularios **HTML**.
- Establecemos la acción del formulario en `{% url 'polls:vote' question.id %}`, y configuramos `method = "post"`. Usar `method = "post"` (en oposición a `method = "get"`) es muy importante, **porque el acto de enviar este formulario alterará los datos del lado del servidor**. Siempre que cree un formulario que altere el lado del servidor de datos, use `method = "post"`. Este consejo no es específico de Django ; es una buena práctica de desarrollo web en general.
- `forloop.counter` indica cuántas veces la etiqueta `for` ha pasado por su bucle
- Dado que estamos creando un formulario **POST** (que puede tener el efecto de modificar los datos), debemos preocuparnos por *las falsificaciones de solicitudes de sitios cruzados*. Afortunadamente, no debemos preocuparnos demasiado, porque Django viene con un sistema útil para protegernos de esto. En resumen, todos los formularios **POST** que están dirigidos a las **URL** internas deben usar la etiqueta de plantilla `{% csrf.token %}`.

Ahora, creemos una vista que maneje los datos enviados y haga algo con ellos. Recuerde, en la parte 3, creamos una **URLconf** para la aplicación de encuestas que incluye esta línea:

```
polls/urls.py

path('<int:question_id>/vote/', views.vote, name='vote'),
```

Figura 36: Agregamos el namespace a `polls/urls.py`

También creamos una implementación ficticia de la función `vote()`. Vamos a crear una versión que sirva de algo. Agreguemos lo siguiente a `polls/views.py`:

```
polls/views.py

from django.http import HttpResponseRedirect, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse

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

Figura 37: Agregamos el namespace a `polls/urls.py`

Este código incluye algunas cosas que aún no hemos cubierto en hasta ahora:

- `request.POST` es un objeto similar a un diccionario que le permite acceder a los datos enviados por nombre de clave. En este caso, `request.POST['choice']` devuelve el **ID** de la opción seleccionada, como una cadena. `request.POST` valores son siempre cadenas.

Tenga en cuenta que Django también proporciona `request.GET` para acceder a los datos **GET** de la misma manera, pero estamos usando explícitamente `request.POST` en nuestro código, para garantizar que los datos solo se modifiquen a través de una llamada **POST**.

- `request.POST['choice']` generará `KeyError` si la elección no se proporcionó en los datos **POST**. El código anterior verifica `KeyError` y vuelve a mostrar el formulario de pregunta con un mensaje de error si no se da la opción.
- Después de incrementar el recuento de opciones, el código devuelve un `HttpResponseRedirect` en lugar de un `HttpResponse` normal. `HttpResponseRedirect` toma un solo argumento: la **URL** a la que se redirigirá al usuario (consulte el siguiente punto para ver cómo construimos la **URL** en este caso).

Como señala el comentario de Python anterior, siempre debe devolver un `HttpResponseRedirect` después de tratar con éxito los datos **POST**. Este consejo no es específico de Django ; es una buena práctica de desarrollo web en general.

- Estamos utilizando la función `reverse()` en el constructor `HttpResponseRedirect` en este ejemplo. Esta función ayuda a evitar tener que codificar una **URL** en la función de vista. Se le da el nombre de la vista a la que

queremos pasar el control y la parte variable del patrón de **URL** que apunta a esa vista. En este caso, usando la **URLconf** que configuramos en la parte 3, esta llamada *reverse()* devolverá una cadena como

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

Donde el **3** es el valor de `question.id`. Esta redirección del URL llamará a la vista *'results'* para mostrar (display) la página final.

Como se mencionó en la parte 3, la solicitud es un objeto `HttpRequest`. Para obtener más información sobre los objetos `HttpRequest`, consulte la [documentación de solicitud y respuesta](#).

Después de que alguien vota en una pregunta, la vista `vote()` redirige a la página de resultados de la pregunta. Escribamos esa vista:

```
polls/views.py

from django.shortcuts import get_object_or_404, render

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

Esto es casi exactamente lo mismo que la vista de *detail()*. La única diferencia es el nombre de la plantilla. Arreglaremos esta redundancia más tarde.

Ahora, creemos una plantilla de resultados en **polls/templates/polls/result.html**

```
polls/templates/polls/results.html

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

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

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

Ahora, vaya a **/polls/1/** en su navegador y vote en la pregunta. Debería ver una página de resultados que se actualiza cada vez que vota. Si envía el formulario sin haber elegido una opción, debería ver un mensaje de error.

## Nota

El código para nuestra vista *vote()* tiene un pequeño problema.

Primero obtiene el objeto `selected_choice` de la base de datos, luego calcula el nuevo valor de los votos y luego lo guarda de nuevo en la base de datos.

Si dos usuarios de su sitio web intentan votar exactamente al mismo tiempo, esto podría salir mal: el mismo valor, digamos 42, se recuperará para los votos.

Luego, para ambos usuarios se calcula y guarda el nuevo valor de 43, pero 44 sería el valor esperado.

Esto se llama condición de carrera. Si está interesado, puede leer [Evitar condiciones de carrera usando F\(\)](#) para aprender cómo puede resolver este problema.

### 3.4.2. Vistas Genéricas (Cuanto menos código, mejor!)

Las vistas *detail()* y *results()* son bastante cortas y además redundantes. La vista *index()*, que muestra una lista de encuestas, es similar.

Estas vistas representan un caso común de desarrollo web básico: obtener datos de la base de datos de acuerdo con un parámetro pasado en la **URL**, cargar una plantilla y devolver la plantilla representada. Como esto es tan común, **Django** proporciona un acceso directo, llamado sistema de “vistas genéricas”.

Las vistas genéricas resumen patrones comunes hasta el punto en que ni siquiera necesita escribir código **Python** para escribir una aplicación.

Convirtamos nuestra aplicación de encuestas para usar el sistema de vistas genéricas, para que podamos eliminar un montón de nuestro propio código. Tendremos que seguir algunos pasos para realizar la conversión:

- Modificar **URLconf**
- Eliminar las viejas vistas
- Introducir las vistas genéricas aportadas por **Django**

#### ¿Porque la evasión de código?

En general, al escribir una aplicación **Django**, evaluará si las vistas genéricas son adecuadas para su problema y las usará desde el principio, en lugar de refactorizar su código a la mitad.

Pero este tutorial se ha centrado intencionalmente en escribir las vistas “por el camino difícil” hasta ahora, para centrarse en los conceptos básicos.

Debe conocer las matemáticas básicas antes de comenzar a usar una calculadora.

#### Modificar URLconf

Abrimos **polls/urls.py** y modificamos los path de la siguiente manera:

```
polls/urls.py

from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
    path('<int:pk>/', views.DetailView.as_view(), name='detail'),
    path('<int:pk>/results/', views.ResultsView.as_view(), name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Notemos que el nombre patrón de la cadena de la ruta tanto del segundo, como del tercer path cambio de **<question\_id>** to **<pk>**.

#### Modificar las vistas

A continuación, vamos a eliminar nuestras vistas antiguas de índice, detalle y resultados y, en su lugar, utilizaremos las vistas genéricas de **Django**. Para hacerlo, abra el archivo **polls/views.py** y cámbielo así:

```

polls/views.py

from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse
from django.views import generic

from .models import Choice, Question

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

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

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

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

def vote(request, question_id):
    ... # same as above, no changes needed.

```

Aquí usamos dos vistas genéricas: `ListView` y `DetailView`. Respectivamente, esas dos vistas resumen los conceptos de “mostrar una lista de objetos” y “mostrar una página de detalles para un tipo particular de objeto”.

- Cada vista genérica necesita saber sobre qué modelo actuará. Esto se proporciona utilizando el atributo `model`.
- La vista genérica `DetailView` espera que el valor de la clave principal capturado de la URL se llame “pk”, por lo que hemos cambiado `question_id` a `pk` para las vistas genéricas.

Por defecto, la vista genérica `DetailView` utiliza una plantilla llamada `<app name>/<model name>_detail.html`. En nuestro caso, usaría la plantilla “`polls/question_detail.html`”. El atributo `template_name` se usa para decirle a Django que use un nombre de plantilla específico en lugar del nombre de plantilla predeterminado autogenerado. También especificamos `template_name` para la vista de la lista de resultados: esto garantiza que la vista de resultados y la vista de detalles tengan una apariencia diferente cuando se procesen, a pesar de que ambas son `DetailView` detrás de escena.

Del mismo modo, la vista genérica `ListView` utiliza una plantilla predeterminada llamada `<app name>/<model name>_list.html`; usamos `template_name` para decirle a `ListView` que use nuestra plantilla existente “`polls/index.html`”.

En partes anteriores del tutorial, las plantillas han recibido un contexto que contiene las variables de contexto `question` y `latest_question_list`. Para `DetailView`, la variable de pregunta se proporciona automáticamente, ya que estamos utilizando un modelo de Django (Pregunta), Django puede determinar un nombre apropiado para la variable de contexto. Sin embargo, para `ListView`, la variable de contexto generada automáticamente es `question_list`. Para anular esto, proporcionamos el atributo `context_object_name`, especificando que queremos usar `latest_question_list` en su lugar. Como un enfoque alternativo, puede cambiar sus plantillas para que coincidan con las nuevas variables de contexto predeterminadas, pero es mucho más fácil decirle a Django que use la variable que desee.

Ejecute el servidor y use su nueva aplicación de sondeo basada en vistas genéricas.

Para obtener detalles completos sobre vistas genéricas, consulte la [documentación de vistas genéricas](#).



## 3.5. Escribiendo tu primera aplicación Django , parte 5

### 3.5.1. Testing: Introduciendo pruebas automatizadas

#### ¿Qué son las pruebas automatizadas?

Los test son rutinas que chequean si tu código opera de manera adecuada.

Las pruebas operan a diferentes niveles. Algunas pruebas pueden aplicarse a un pequeño detalle (*¿un método de modelo en particular devuelve los valores esperados?*), mientras que otras examinan el funcionamiento general del software (*¿produce una secuencia de entradas del usuario en el sitio el resultado deseado?*). Eso no es diferente del tipo de prueba que realizó anteriormente en la segunda parte, utilizando el *shell* para examinar el comportamiento de un método, o ejecutando la aplicación e ingresando datos para verificar cómo se comporta.

La diferencia en las pruebas automatizadas es que el sistema realiza el trabajo de prueba por vos. Podes crear un conjunto de pruebas una vez, y después, a medida que realizas cambios en tu aplicación, puedes verificar que tu código siga funcionando como querías , sin tener que realizar pruebas manuales que requieren mucho tiempo.

#### ¿Porque tenemos que hacer tests?

Quizas sientas que ya tenes demasiado aprendiendo **Python/Django**, y tener otra cosa más que aprender y hacer puede parecer abrumador y quizás innecesario. Después de todo, nuestra aplicación de encuestas está funcionando bastante bien ahora; pasar por el problema de crear pruebas automatizadas no va a hacer que funcione mejor. Si crear la aplicación de encuestas es la última parte de la programación de Django que harás, entonces es cierto, no necesitas saber cómo crear pruebas automatizadas. Pero, si ese no es el caso, ahora es un excelente momento para aprender.

#### *Los Test pueden ahorrarte tiempo*

Hasta cierto punto, “comprobar que parece funcionar” será una prueba satisfactoria. En una aplicación más sofisticada, puede tener docenas de interacciones complejas entre componentes.

Un cambio en cualquiera de esos componentes podría tener consecuencias inesperadas en el comportamiento de la aplicación. Verificar que todavía “parece funcionar” podría significar revisar la funcionalidad de su código con veinte variaciones diferentes de sus datos de prueba para asegurarse de que no ha roto algo, no es un buen uso de su tiempo.

Eso es especialmente cierto cuando las pruebas automatizadas pueden hacer esto por usted en segundos. Si algo sale mal, las pruebas también ayudarán a identificar el código que está causando el comportamiento inesperado.

A veces puede parecer una tarea difícil separarse de su trabajo productivo y creativo de programación para enfrentar el negocio poco atractivo y poco emocionante de escribir pruebas, particularmente cuando sabe que su código funciona correctamente.

Sin embargo, la tarea de escribir pruebas es mucho más satisfactoria que pasar horas probando su aplicación manualmente o tratando de identificar la causa de un problema recientemente introducido.

#### *Los test no solo identifican problemas, los previenen*

Es un error pensar en las pruebas simplemente como un aspecto negativo del desarrollo.

Sin pruebas, el propósito o el comportamiento previsto de una aplicación podría ser bastante opaco. Incluso cuando se trata de su propio código, a veces se encontrará hurgando en él tratando de averiguar qué está haciendo exactamente.

Las pruebas cambian eso; iluminan su código desde adentro, y cuando algo sale mal, enfocan la luz en la parte que salió mal, incluso si ni siquiera se había dado cuenta de que había salido mal.

#### *Los testeos hacen que tu código sea mas atractivo*

Es posible que haya creado un software brillante, pero encontrará que muchos otros desarrolladores se negarán a verlo porque carece de pruebas; sin pruebas, no confiarán en ello. Jacob Kaplan-Moss, uno de los desarrolladores originales de Django, dice que <sup>el</sup> código sin pruebas está roto por diseño”.

Que otros desarrolladores quieran ver pruebas en su software antes de que lo tomen en serio es otra razón más para que comience a escribir pruebas.

## *Los testeos son de mucha ayuda a la hora de trabajar en equipo*

Los puntos anteriores están escritos desde el punto de vista de un único desarrollador que mantiene una aplicación. Las aplicaciones complejas serán mantenidas por los equipos. Las pruebas garantizan que los colegas no rompan accidentalmente su código (y que no lo haga sin saberlo). Si quiere ganarse la vida como programador de Django, ¡debe ser bueno escribiendo pruebas!

### 3.5.2. Estrategias básicas de Testing

Hay muchas formas de abordar las pruebas de escritura.

Algunos programadores siguen una disciplina llamada [Desarrollo controlado de test \(test-driven development\)](#); en realidad escriben sus pruebas antes de escribir su código. Esto puede parecer contrario a la intuición, pero de hecho es similar a lo que la mayoría de la gente suele hacer de todos modos: describen un problema y luego crean un código para resolverlo. El desarrollo basado en pruebas formaliza el problema en un caso de prueba de Python.

Más a menudo, un recién llegado a las pruebas creará un código y luego decidirá que debe tener algunas pruebas. Tal vez hubiera sido mejor escribir algunas pruebas antes, pero nunca es demasiado tarde para comenzar.

A veces es difícil descubrir por dónde empezar a escribir exámenes. Si ha escrito varios miles de líneas de Python, elegir algo para probar podría no ser fácil. En tal caso, es fructífero escribir su primera prueba la próxima vez que realice un cambio, ya sea cuando agregue una nueva función o repare un error.

### 3.5.3. Escribiendo nuestro primer Test

#### Identificando Bugs

Afortunadamente, hay un pequeño error en la aplicación de encuestas que podemos solucionar de inmediato: el método `Question.was_published_recently()` devuelve `True` si la pregunta se publicó en el último día (lo cual es correcto) pero también si el campo `pub_date` de la pregunta está en el futuro (que ciertamente no lo es).

Confirme el error utilizando el *shell* para verificar el método en una pregunta cuya fecha se encuentra en el futuro:

```
$ python manage.py shell

>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() +
datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

Como las cosas en el futuro no son recientes”, esto es claramente incorrecto.

#### Creamos un Test para exponer al Bug

Lo que acabamos de hacer en el *shell* para probar el problema es exactamente lo que podemos hacer en una prueba automatizada, así que vamos a convertir eso en una prueba automatizada.

Un lugar convencional para las pruebas de una aplicación es en el archivo `tests.py` de la aplicación; el sistema de prueba encontrará pruebas automáticamente en cualquier archivo cuyo nombre comience con prueba.

Ponga lo siguiente en el archivo `tests.py` en la aplicación de encuestas:

polls/tests.py

```
import datetime

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

from .models import Question

class QuestionModelTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() returns False for questions whose pub_date
        is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

Aquí hemos creado una subclase `django.test.TestCase` con un método que crea una instancia de `Question` con `pub_date` en el futuro. Luego verificamos el resultado de `was_published_recently()`, que debería ser falso.

#### 3.5.4. Corriendo Testeos

Podemos correr nuestro test desde la terminal:

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

Y vas a ver algo como:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
=====
FAIL: test_was_published_recently_with_future_question
(polls.tests.QuestionModelTests)
-----
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False
-----
Ran 1 test in 0.001s

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

## ¿Tienes un error diferente?

Si, en cambio, está obteniendo un `NameError` aquí, es posible que haya omitido un paso en la *Parte 2* donde agregamos las importaciones de *fecha y hora* y *zona horaria* a `polls/models.py`. Copie las importaciones de esa sección e intente ejecutar sus pruebas nuevamente.

Lo que paso es esto:

- `manage.py test polls` busco el archivo `tests.py` en la aplicación `polls`.
- Este encontró una subclase `django.test.TestCase`
- Este crea una base de datos adicional para el propósito del test.
- Este busca los metodos de testeos, los cuales son todos los métodos que empiezan con `test`.
- En `test_was_published_recently_with_future_question` se crea una instancia de `Question` cuyo campo `pub_date` estará unos 30 días en el futuro respecto del día presente.
- ... y usando el método `assertIs()`, este descubre que `was_published_recently()` devuelve `RTrue`, sabiendo que queríamos que este devuelva `RFalse`.

La prueba nos informa qué prueba falló e incluso la línea en la que ocurrió la falla.

### 3.5.5. Arreglamos el Bug

Ya sabemos cuál es el problema: `Question.was_published_recently()` debería devolver `False` si su `pub_date` está en el futuro. Modifiquemos el método en `models.py`, para que solo devuelva `True` si la fecha también está en el pasado:

`polls/models.py`

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

Corriendo otra vez el comando del testeo, deberíamos ver lo siguiente:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----
Ran 1 test in 0.001s

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

Entonces, lo que hicimos fue identificar un error mediante el test que escribimos, luego corregimos el error que en este caso estaba en la método `Question.was_published_recently()` dentro del modelo `Question`, cuando este pasa el test el modelo esta correctamente escrito.

### 3.5.6. Tests más completos

Mientras estamos aquí, podemos precisar el método `was_published_recently()`; de hecho, sería vergonzoso si al corregir un error hubiéramos introducido otro.

Agregue dos métodos de prueba más a la misma clase, para probar el comportamiento del método de manera más integral:

```
polls/tests.py

def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() returns False for questions whose pub_date
    is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose pub_date
    is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59,
seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(), True)
```

Y ahora tenemos tres pruebas que confirman que `Question.was_published_recently()` devuelve valores razonables para preguntas pasadas, recientes y futuras.

Una vez más, las encuestas son una aplicación mínima, pero por compleja que sea en el futuro y cualquier otro código con el que interactúe, ahora tenemos alguna garantía de que el método para el que hemos escrito las pruebas se comportará de la manera esperada.

### 3.5.7. Testeando vistas

La aplicación de encuestas es bastante indiscriminada: publicará cualquier pregunta, incluidas aquellas cuyo campo `pub.date` se encuentre en el futuro. Deberíamos mejorar esto. Establecer una `pub.date` en el futuro debería significar que la Pregunta se publica en ese momento, pero invisible hasta ese momento.

#### Tests para vistas

Cuando solucionamos el error anterior, primero escribimos la prueba y luego el código para solucionarlo. De hecho, ese fue un ejemplo de desarrollo basado en pruebas, pero en realidad no importa en qué orden hagamos el trabajo.

En nuestra primera prueba, nos centramos estrechamente en el comportamiento interno del código. Para esta prueba, queremos verificar su comportamiento, ya que lo experimentaría un usuario a través de un navegador web.

Antes de intentar arreglar cualquier cosa, echemos un vistazo a las herramientas a nuestra disposición.

#### El cliente de pruebas de Django

Django tiene un cliente de testeo para simular con interactua el usuario con el código al nivel de las vistas. Podemos usarlo mediante el `shell` o `tests.py`

Comenzaremos nuevamente con el `shell`, donde debemos hacer un par de cosas que no serán necesarias en `tests.py`. El primero es configurar el entorno de prueba en el `shell`:

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

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

`setup_test_environment()` instala un renderizador de plantillas que nos permitirá examinar algunos atributos adicionales en las respuestas, como `response.context` que de otro modo no estaría disponible. Tenga en cuenta que este método no configura una base de datos de prueba, por lo que lo siguiente se ejecutará en la base de datos existente y el resultado puede diferir ligeramente dependiendo de las preguntas que ya haya creado. Es posible que obtenga resultados inesperados si su `TIME_ZONE` en `settings.py` no es correcta. Si no recuerda haberlo configurado antes, verifíquelo antes de continuar.

A continuación, necesitamos importar la clase de cliente de prueba (más adelante en `tests.py` usaremos la clase `django.test.TestCase`, que viene con su propio cliente, por lo que no será necesario):

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

Con esto listo, podemos decirle al cliente que haga algunos trabajos para nosotros:

```
>>> # get a response from '/'
>>> response = client.get('/')
Not Found: /
>>> # we should expect a 404 from that address; if you instead see an
>>> # "Invalid HTTP_HOST header" error and a 400 response, you probably
>>> # omitted the setup_test_environment() call described earlier.
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n  <ul>\n    \n      <li><a href="/polls/1/">What's up?</a>
</li>\n  \n  </ul>\n\n'
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>>>
```

Getting H

## Mejorando nuestra vista

La lista de encuestas muestra encuestas que aún no se han publicado (es decir, aquellas que tengan una fecha de publicación en el futuro). Vamos a arreglar eso.

En la parte 4 presentamos una vista basada en clases, basada en **ListView**:

```
polls/views.py

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

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

Debemos corregir el `get_queryset()` y cambiarlo para que también verifique la fecha comparándola con `timezone.now()`. Primero necesitamos agregar una importación y luego corregir dicho método:

```
polls/views.py

from django.utils import timezone

polls/views.py

def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

`Question.objects.filter(pub_date__lte = timezone.now ())` devuelve un conjunto de consultas que contiene preguntas cuyo `pub_date` es menor o igual a, es decir, anterior o igual a `timezone.now`.

### Testeamos nuestra vista

Ahora puedes asegurarte de que esto se comporta como esperas al activar el servidor de ejecución, cargar el sitio en su navegador, crear preguntas con fechas en el pasado y en el futuro, y verificar que solo se enumeren los que se han publicado. Esto es medio embole, así que también vamos a crear una prueba basada en nuestra sesión de *shell* anterior.

Agreguemos lo siguiente al `tests.py`:

```
polls/tests.py

from django.urls import reverse

def create_question(question_text, days):
    """
    Create a question with the given `question_text` and published the
    given number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text,
        pub_date=time)
```



```

class QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_past_question(self):
        """
        Questions with a pub_date in the past are displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_future_question(self):
        """
        Questions with a pub_date in the future aren't displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-30)
        create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>', '<Question: Past question 1.>']
        )

```

A ver, tratemos de entender esto.

La primera es una función de acceso directo a preguntas, `create_question`, se usa para tomar algunas repeticiones del proceso de creación de preguntas.

Por otro lado, `test_no_questions` no crea ninguna pregunta, pero comprueba el mensaje: "No hay encuestas disponibles". y verifica que la última lista de preguntas está vacía. Tenga en cuenta que la clase `django.test.TestCase` proporciona algunos métodos de aserción adicionales. En estos ejemplos, utilizamos `assertContains()` y `assertQuerysetEqual()`.



En `test_past_question`, creamos una pregunta y verificamos que aparezca en la lista.

En `test_future_question`, creamos una pregunta con `pub_date` en el futuro. La base de datos se restablece para cada método de prueba, por lo que la primera pregunta ya no está allí, y nuevamente el índice no debería tener ninguna pregunta.

Y así. En efecto, estamos utilizando las pruebas para contar una historia de la entrada del 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.

### Testeamos el `DetailView`

Lo que tenemos funciona bien; sin embargo, aunque las preguntas futuras no aparezcan en el índice, los usuarios aún pueden comunicarse con ellas si conocen o adivinan la **URL** correcta. Por lo tanto, debemos agregar una restricción similar a `DetailView`:

```
polls/views.py

class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

Agregaremos algunas pruebas, para verificar que una Pregunta cuya `pub_date` esté en el pasado se pueda mostrar, y que una con `pub_date` en el futuro no:

```
polls/tests.py

class QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns a 404 not found.
        """
        future_question = create_question(question_text='Future question.', days=-5)

        url = reverse('polls:detail', args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_past_question(self):
        """
        The detail view of a question with a pub_date in the past
        displays the question's text.
        """
        past_question = create_question(question_text='Past Question.', days=-5)

        url = reverse('polls:detail', args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

### 3.5.8. Algunas ideas para la creación de otros tests

Deberíamos agregar un método `get_queryset` similar a `ResultsView` y crear una nueva clase de prueba para esa vista. Será muy similar a lo que acabamos de crear; de hecho habrá mucha repetición.

También podríamos mejorar nuestra aplicación de otras maneras, agregando pruebas en el camino. Por ejemplo, es una tontería que las preguntas se puedan publicar en el sitio que no tiene opciones. Por lo tanto, nuestras opiniones podrían verificar esto y excluir tales Preguntas. Nuestras pruebas crearían una pregunta sin opciones y luego comprobarían que no está publicada, así como crearían una pregunta similar con opciones y comprobarían que está publicada.

Tal vez a los usuarios administradores registrados se les debería permitir ver preguntas no publicadas, pero no visitantes comunes. Una vez más: lo que sea necesario agregar al software para lograr esto debe ir acompañado de una prueba, ya sea que escriba la prueba primero y luego haga que el código pase la prueba, o calcule primero la lógica en su código y luego escriba una prueba.

En cierto momento, seguramente verás tus pruebas y te preguntarás si tu código sufre de hinchazón de prueba, lo que nos lleva a:

#### Cuando hacemos pruebas, más es mejor

Puede parecer que nuestras pruebas están fuera de control. A este ritmo, pronto habrá más código en nuestras pruebas que en nuestra aplicación, y la repetición no es estética, en comparación con la elegante concisión del resto de nuestro código.

No importa. Déjalos crecer. En su mayor parte, puede escribir una prueba una vez y luego olvidarse de ella. Continuará realizando su útil función a medida que continúe desarrollando su programa.

Algunas veces las pruebas deberán actualizarse. Supongamos que modificamos nuestras opiniones para que solo se publiquen Preguntas con opciones. En ese caso, muchas de nuestras pruebas existentes fallarán, diciéndonos exactamente qué pruebas deben modificarse para actualizarlas, por lo que las pruebas ayudan a cuidarse a sí mismas.

En el peor de los casos, a medida que continúe desarrollando, es posible que tenga algunas pruebas que ahora son redundantes. Incluso eso no es un problema; en la prueba de redundancia es algo bueno.

Mientras sus exámenes estén organizados de manera sensata, no serán inmanejables. Las buenas reglas generales incluyen tener:

- un `TestClass` separado para cada modelo o vista
- un método de prueba separado para cada conjunto de condiciones que desea probar
- nombres de métodos de prueba que describen su función

#### Pruebas adicionales

Este tutorial solo presenta algunos de los conceptos básicos de las pruebas. Hay mucho más que puede hacer y una serie de herramientas muy útiles a su disposición para lograr algunas cosas muy inteligentes.

Por ejemplo, si bien nuestras pruebas aquí han cubierto parte de la lógica interna de un modelo y la forma en que nuestras vistas publican información, puede usar un marco <sup>en</sup> el navegador como `Selenium` para probar la forma en que su **HTML** se procesa en un navegador. Estas herramientas le permiten verificar no solo el comportamiento de su código `Django`, sino también, por ejemplo, de su **JavaScript**. ¡Es bastante ver que las pruebas inician un navegador y comienzan a interactuar con su sitio, como si un ser humano lo condujera! `Django` incluye `LiveServerTestCase` para facilitar la integración con herramientas como `Selenium`.

Si tiene una aplicación compleja, es posible que desee ejecutar pruebas automáticamente con cada confirmación con el fin de una [integración continua](#), de modo que el control de calidad sea en sí mismo, al menos parcialmente, automatizado.

Una buena manera de detectar partes no probadas de su aplicación es verificar la cobertura del código. Esto también ayuda a identificar el código frágil o incluso muerto. Si no puede probar un fragmento de código, generalmente significa que el código debe ser refactorizado o eliminado. La cobertura ayudará a identificar el código muerto. Vea [Integración con coverage.py](#) para más detalles.

Las pruebas en `Django` ([Testing in Django](#)) tienen información completa sobre las pruebas.

### 3.6. Escribiendo tu primera aplicación Django , parte 6

Ya hemos creado el proyecto, la aplicación de encuesta y todo el back de la página y ahora agregaremos una hoja de estilo y una imagen, para tener un poco más de front.

Además del **HTML** generado por el servidor, las aplicaciones web generalmente necesitan servir archivos adicionales, como imágenes, **JavaScript** o **CSS**, necesarios para representar la página web completa. En **Django** , nos referimos a estos archivos como "archivos estáticos".

Para proyectos pequeños, esto no es un gran problema, ya que puede mantener los archivos estáticos en algún lugar donde su servidor web pueda encontrarlos. Sin embargo, en proyectos más grandes, especialmente aquellos compuestos por múltiples aplicaciones, tratar con los múltiples conjuntos de archivos estáticos proporcionados por cada aplicación comienza a ser complicado.

Para eso es `django.contrib.staticfiles`: recopila archivos estáticos de cada una de sus aplicaciones (y de cualquier otro lugar que especifique) en una única ubicación que se pueda servir fácilmente en producción.

#### 3.6.1. Personaliza el aspecto de tu aplicación

Primero, cree un directorio llamado `static` en su directorio de encuestas. Django buscará archivos estáticos allí, de manera similar a cómo Django encuentra plantillas dentro de encuestas / templates /.

La configuración `STATICFILES_FINDERS` de Django contiene una lista de buscadores que saben cómo descubrir archivos estáticos de varias fuentes. Uno de los valores predeterminados es `AppDirectoriesFinder`, que busca un subdirectorio ".estático" en cada una de las `INSTALLED_APPS`, como el de las encuestas que acabamos de crear. El sitio de administración utiliza la misma estructura de directorio para sus archivos estáticos.

Dentro del directorio estático que acaba de crear, cree otro directorio llamado `encuestas` y dentro de ese, cree un archivo llamado `style.css`. En otras palabras, su hoja de estilo debe estar en `polls/static/polls/style.css`. Debido a cómo funciona el buscador de archivos estáticos `AppDirectoriesFinder`, puede referirse a este archivo estático en Django como `polls/style.css`, similar a cómo hace referencia a la ruta de las plantillas.

#### static file namespacing

Al igual que las plantillas, podríamos evitar poner nuestros archivos estáticos directamente en `polls/static` (en lugar de crear otro subdirectorio de encuestas), pero en realidad sería una mala idea.

Django elegirá el primer archivo estático que encuentre cuyo nombre coincida, y si tuviera un archivo estático con el mismo nombre en una aplicación diferente, Django no podría distinguirlos.

Necesitamos poder apuntar a Django al correcto, y la mejor manera de asegurar esto es espaciando los nombres.

Es decir, al poner esos archivos estáticos dentro de otro directorio nombrado para la aplicación misma.

Ponga el siguiente código en esa hoja de estilo (`polls/static/polls/style.css`):

```
polls/static/polls/style.css
```

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

Luego, agregamos lo siguiente a `polls/templates/polls/index.html`:

```
polls/templates/polls/index.html
```

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

La etiqueta de plantilla `{% static %}` genera la **URL** absoluta de los archivos estáticos. Eso es todo lo que necesitas hacer para el desarrollo. Inicie el servidor (o reinícielo)

### 3.6.2. Imagen de fondo

A continuación, crearemos un subdirectorio para imágenes. Cree un subdirectorio de imágenes en el directorio encuestas `/static/polls/`. Dentro de este directorio, coloque una imagen llamada `background.gif`. En otras palabras, coloque su imagen en encuestas `/static/polls/images/background.gif`.

Luego, agregue a su hoja de estilo (`polls/static/polls/style.css`):

```
polls/static/polls/style.css
```

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

Luego, reinicia el navegador.

## Cuidado!

Por supuesto, la etiqueta de plantilla `{% static %}` no está disponible para su uso en archivos estáticos como su hoja de estilo que no son generados por **Django**.

Siempre debe usar rutas relativas para vincular sus archivos estáticos entre sí, porque luego puede cambiar `STATIC_URL` (utilizado por la etiqueta de plantilla `estática` para generar sus URL) sin tener que modificar también un montón de rutas en sus archivos estáticos.

Estos son los conceptos básicos. Para obtener más detalles sobre la configuración y otros bits incluidos en el marco, consulte [los procedimientos de archivos estáticos](#) y [la referencia de archivos estáticos](#). La [implementación de archivos estáticos](#) explica cómo usar archivos estáticos en un servicio real.

## 4. Referencias

### 4.1. Refes de las partes de los tutoriales

- [Parte 1](#)
- [Parte 2](#)
- [parte 3](#)
- [parte 4](#)
- [parte 5](#)
- [parte 6](#)
- [parte 7](#)
- [Reusable Apps](#)

### 4.2. Refes W3School

- [Html](#)
- [css](#)
- [javascript](#)
- [Python](#)
- [Bootstrap](#)

### 4.3. Atajos

- [Crear entorno de python con conda](#)
- [Cambiar de version de python con conda](#)
- [LANGUAGE\\_CODE](#)
- [TIME\\_ZONE](#)