Django

¿Qué es Django?

Django es un framework web de alto nivel escrito en Python.

Facilita el desarrollo rápido de aplicaciones web seguras y mantenibles.

Es gratuito y de código abierto1.

Características Principales

Desarrollo Rápido:

Permite pasar de la idea al producto final rápidamente.

Sigue la filosofía de “baterías incluidas”, proporcionando muchas funcionalidades listas para usar2.

Seguridad:

Ayuda a los desarrolladores a evitar errores comunes de seguridad.

Incluye protección contra ataques como SQL injection, cross-site scripting (XSS), y cross-site request forgery (CSRF)2.

Escalabilidad:

Utilizado por algunos de los sitios web más concurridos del mundo.

Facilita la escalabilidad tanto en términos de tráfico como de complejidad de la aplicación1.

Componentes Principales

Modelos (Models): Definen la estructura de la base de datos.

Vistas (Views): Controlan la lógica de la aplicación y la interacción con los modelos.

Plantillas (Templates): Generan el HTML que se envía al navegador del usuario.

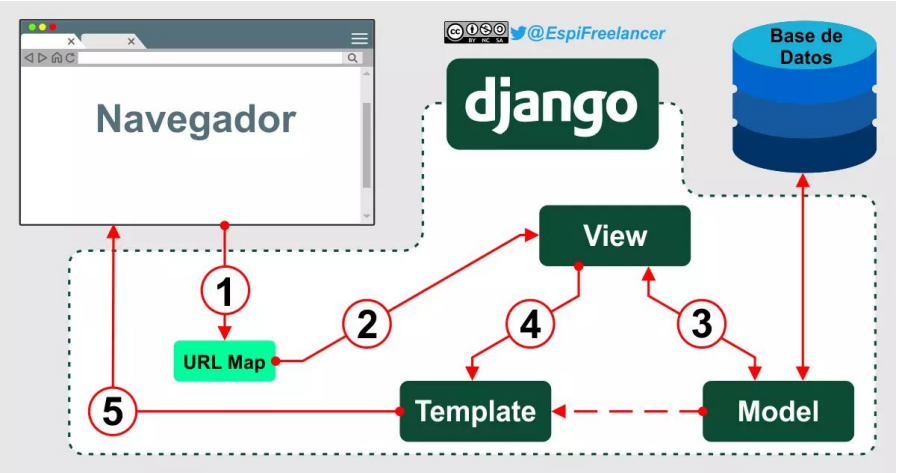
URLs: Mapean las URLs a las vistas correspondientes2.

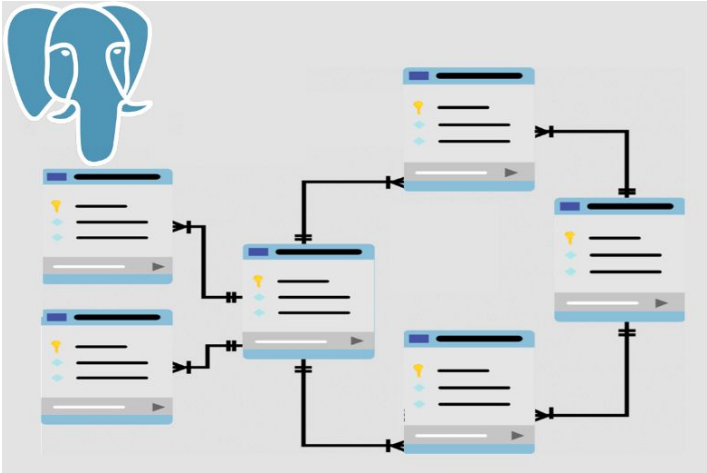
Diseño y Arquitectura

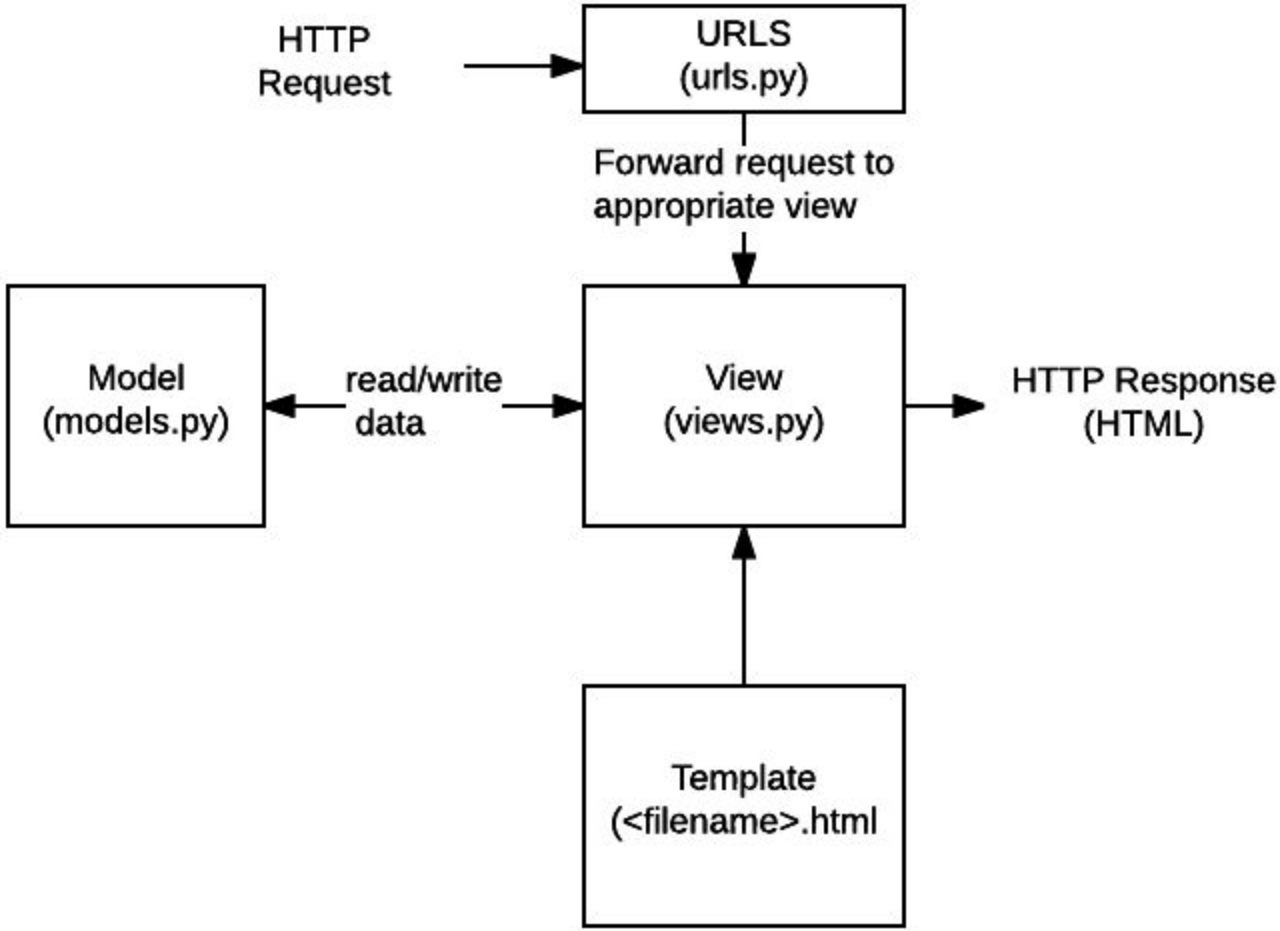
Aplicación Web

Base de datos Relacional (postgresql)

[Model-View-Template Architecture]






HTTP
Request
URLS
(urls.py)
Forward request to
appropriate view
Model
(models.py)
read/write
data
View
(views.py)
HTTP Response
(HTML)
Template
(<filename>.html

Preparación del entorno de desarrollo

PYTHON

- PIP
- VIRTUALENV

PROYECTO DJANGO

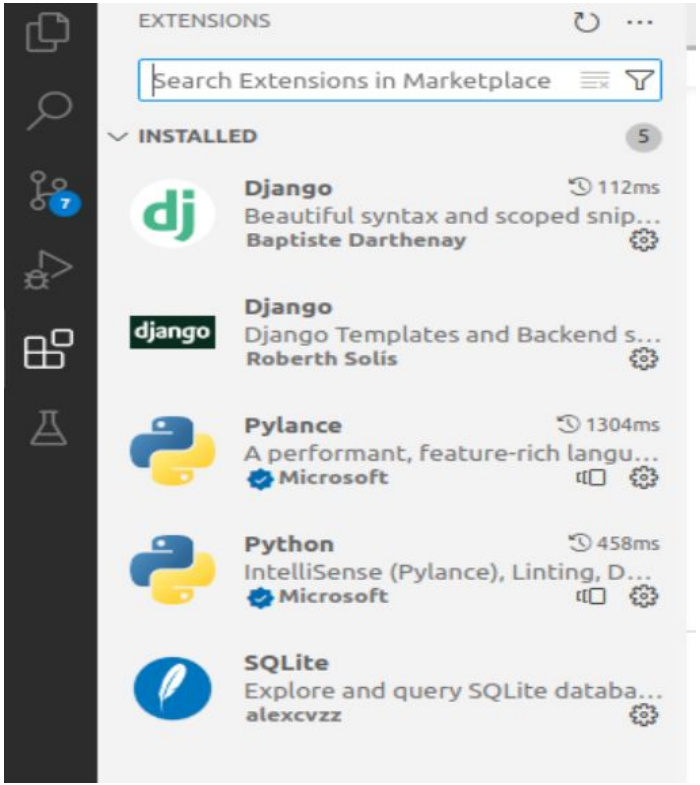
- .gitignore
- requeriments.txt
- ENV
- postgresql

Preparación del entorno de desarrollo

GIT

VSCODE

- Versionar con las herramienta del IDE
- Extensiones
- Run and Debug



Creando un proyecto

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

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

El directorio raíz externo mysite/es un contenedor para tu proyecto. Su nombre no es importante para Django; puedes cambiarle el nombre como quieras.

manage.py:Una utilidad de línea de comandos que te permite interactuar con este proyecto de Django de varias maneras. Puedes leer todos los detalles en manage.pydjango -admin y manage.py .

El mysite/directorio interno es el paquete Python real para tu proyecto. Su nombre es el nombre del paquete Python que necesitarás usar para importar cualquier cosa dentro de él (por ejemplo, mysite.urls).

mysite/__init__.py: Un archivo vacío que le indica a Python que este directorio debe considerarse un paquete de Python. Si eres principiante en Python, lee más sobre los paquetes en la documentación oficial de Python.

Creando un proyecto

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

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

mysite/settings.py: Configuración de este proyecto de Django. La configuración de Django le informará sobre cómo funcionan las configuraciones.

mysite/urls.py: Las declaraciones de URL para este proyecto de Django; una “tabla de contenidos” de su sitio basado en Django. Puede leer más sobre las URL en el despachador de URL .

mysite/asgi.py: Un punto de entrada para servidores web compatibles con ASGI que sirvan a su proyecto. Consulte Cómo implementar con ASGI para obtener más detalles.

mysite/wsgi.py: Un punto de entrada para servidores web compatibles con WSGI para brindar servicio a su proyecto. Consulte Cómo implementar con WSGI para obtener más detalles.

Servidor de desarrollo

Comprobemos que su proyecto Django funciona. Cambie al directorio externo **`mysite`**, si todavía no lo ha hecho, y ejecute los siguientes comandos:

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

Ahora que el servidor está funcionando, visita http://127.0.0.1:8000/ con tu navegador web. Verás una página de «¡Felicitaciones!» con un cohete despegando. ¡Funcionó!

Ha iniciado el servidor de desarrollo de Django, un servidor web liviano escrito exclusivamente en Python. Lo hemos incluido con Django para que pueda desarrollar cosas rápidamente, sin tener que lidiar con la configuración de un servidor de producción (como Apache) hasta que esté listo para la producción.

No utilice este servidor en nada que se parezca a un entorno de producción. Está destinado únicamente a utilizarse durante el desarrollo.

Recarga automática del comando `runserver`

Creando aplicaciones (App)

Cada aplicación que usted escribe en Django consiste en un paquete de Python que sigue una determinada convención. Django tiene una utilidad que genera automáticamente la estructura básica de directorios de una aplicación.

Tus aplicaciones pueden estar en cualquier parte de tu ruta de Python .

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 blog, 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 varias aplicaciones. Una aplicación puede estar en varios proyectos.

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

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

Esto creará un directorio **`polls`**, que se presenta de la siguiente manera:

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

Esta estructura de directorios almacenará la aplicación encuesta.

Escriba su primera vista (views)

polls/views.py ¶

```python
from django.http import HttpResponse


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

1) Esta es la vista más básica posible en Django. Para acceder a ella en un navegador, necesitamos asignarla a una URL, y para ello necesitamos definir una configuración de URL, o «URLconf» para abreviar.

polls/urls.py ¶

```python
from django.urls import path

from . import views

urlpatterns = [
    path("", views.index, name="index"),
]
```

2) Estas configuraciones de URL se definen dentro de cada aplicación de Django y son archivos Python llamados `urls.py`.

Escriba su primera vista (views)

3) El siguiente paso es configurar la URLconf global en el proyecto `mysite` para incluir la URLconf definida en `polls.urls`. Para ello, agregue una importación para `django.urls.include` en `mysite/urls.py` e inserte un `include()` en la lista `urlpatterns`, de modo que tenga:

mysite/urls.py ¶

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

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

La función `include()` permite hacer referencia a otros URLconfs. Cada vez que Django encuentra `include()` corta cualquier parte de la URL que coincide hasta ese punto y envía la cadena restante a la URLconf incluida para seguir el proceso.

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

Escriba su primera vista (views)

Vaya a http://localhost:8000/polls/ en su navegador, y usted debería ver el texto «*Hello, world. You're at the polls index.*» el cual definió en la vista `index`.

La `path()` función recibe cuatro argumentos, dos requeridos `route` y `view`; y dos opcionales `kwargs` y `name`. Este es el momento de revisar para que sirven estos argumentos.

`route`

es una cadena que contiene un patrón de URL. Cuando Django procesa una petición comienza por el primer patrón en `urlpatterns` y continua hacia abajo por la lista comparando la URL solicitada con cada patrón hasta encontrar aquel que calza.

`view`¶

Cuando Django encuentra una coincidencia de expresiones regulares llama a la función de la vista especificada con un objeto `HttpRequest` como primer argumento y cualquiera de los valores «capturados» de la ruta como argumentos de palabra clave. Le daremos un ejemplo de esto en un momento.

`kwargs`¶

Los argumentos arbitrarios de palabra clave se pueden pasar en un diccionario a la vista destino. No vamos a utilizar esta funcionalidad de Django en el tutorial.

`name`¶

Dar un nombre a su URL le permite referirse a ella de forma inequívoca desde otras partes de Django sobre todo desde las plantillas. Esta potente característica le permite realizar cambios globales en los patrones de URL de su proyecto modificando solo un único archivo.

Configuración de la base de datos

Ahora, abre **`mysite/settings.py`**. Es un módulo Python normal con variables a nivel de módulo que representan configuraciones de Django.

De forma predeterminada, la **`DATABASES`** configuración utiliza SQLite.

Mientras esté editando **`mysite/settings.py`**, configúrelo **`TIME_ZONE`** en su zona horaria.

tenga en cuenta la **`INSTALLED_APPS`**configuración que aparece en la parte superior del archivo. Esta contiene los nombres de todas las aplicaciones de Django que están activadas en esta instancia de Django. Las aplicaciones se pueden usar en varios proyectos y usted puede empaquetarlas y distribuirlas para que otras personas las usen en sus proyectos. De forma predeterminada, **`INSTALLED_APPS`**contiene las siguientes aplicaciones, todas las cuales vienen con Django:

- **`django.contrib.admin`**– El sitio de administración. Lo usarás en breve.
- **`django.contrib.auth`**– Un sistema de autenticación.
- **`django.contrib.contenttypes`**– Un marco para tipos de contenido.
- **`django.contrib.sessions`**– Un marco de sesión.
- **`django.contrib.messages`**– Un marco de mensajería.
- **`django.contrib.staticfiles`**– Un marco para gestionar archivos estáticos.

Configuración de la base de datos

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 ello, ejecute el siguiente comando:

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

El **`migrate`** comando analiza la **`INSTALLED_APPS`** configuración y crea las tablas de base de datos necesarias según la configuración de la base de datos en su **`mysite/settings.py`** archivo y las migraciones de base de datos enviadas con la aplicación.

Creando modelos ¶

Definiremos sus modelos: esencialmente, el diseño de su base de datos, con metadatos adicionales.

En la aplicación de encuestas, crearemos dos modelos: **`Question`** y **`Choice`**. A **`Question`** tiene una pregunta y una fecha de publicación. A **`Choice`**tiene dos campos: el texto de la elección y un recuento de votos. Cada uno **`Choice`** está asociado con un **`Question`**.

polls/models.py ¶

```python
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)
```

Activación de modelos ¶

Ese pequeño fragmento de código de modelo le proporciona a Django mucha información. Con él, Django puede:

- Cree un esquema de base de datos (declaraciones) para esta aplicación.**`CREATE TABLE`**
- Cree una API de acceso a bases de datos de Python para acceder **`Question`**a **`Choice`**objetos.
-

Primero debemos decirle a nuestro proyecto que la `polls` aplicación está instalada.

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",
]
```

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

/

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

Activación de modelos ¶

Hay un comando que ejecutará las migraciones por usted y administrará su esq
de base de datos automáticamente; se llama **`migrate`**, y lo abordaremos en un
momento; pero primero, veamos qué SQL ejecutaría esa migración. El
`sqlmigrate`comando toma los nombres de las migraciones y devuelve su SQL

/

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

mysite/settings.py ¶

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

Jugando con la API ¶

Ahora, entremos en el shell interactivo de Python y experimentemos con la API gratuita que te ofrece Django. Para invocar el shell de Python, usa este comando: `$ python manage.py shell`

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

Una vez que esté en el shell, explore la API de la base de datos :

Practicar

```
>>> 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=datetime.timezone.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()
```

Presentando el administrador de Django

Creando un usuario administrador ¶

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

/

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

, inícielo de la siguiente manera:

/

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

Ahora, abre un navegador web y ve a “/admin/” en tu dominio local, por ejemplo, http://127.0.0.1:8000/admin/ . Deberías ver la pantalla de inicio de sesión del administrador:

Presentando el administrador de Django

Hacer que la aplicación de encuesta sea modificable en el administrador ¶

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

Solo queda una cosa por hacer: debemos indicarle al administrador que `Question`los objetos tienen una interfaz de administración. Para ello, abra el `polls/admin.py`archivo y edítelo para que se vea así:

polls/admin.py ¶

```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

Escribir tu primera aplicación Django

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

- Página de inicio del blog: muestra las últimas entradas.
- Página de “detalle” de entrada: página de enlace permanente para una sola entrada.
- Página de archivo basada en año: muestra todos los meses con entradas en el año determinado.
- Página de archivo basada en meses: muestra todos los días con entradas en el mes determinado.
- Página de archivo basada en días: muestra todas las entradas del día determinado.
- Acción de comentario: maneja la publicación de comentarios en una entrada determinada.

Escribir tu primera aplicación Django

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

- Página de “índice” de preguntas: muestra las últimas preguntas.
- Página de “detalle” de la pregunta: muestra un texto de pregunta, sin resultados pero con un formulario para votar.
- Página de “resultados” de la pregunta: muestra los resultados de una pregunta en particular.
- Acción de votación: maneja la votación para una opción 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 un método, en el caso de las 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).

Un patrón de URL es la forma general de una URL, por ejemplo: **`/newsarchive/<year>/<month>/`**.

Para llegar desde una URL a una vista, Django utiliza lo que se conoce como "URLconfs". Una URLconf asigna patrones de URL a vistas.

Escribir tu primera aplicación Django (Views)

Escribiendo más vistas ¶

Ahora, agreguemos algunas vistas más a `polls/views.py`. Estas vistas son ligeramente diferentes, porque toman un argumento

Conecte estas nuevas vistas al `polls.urls` módulo agregando las siguientes `path()` llamadas:

polls/views.py ¶

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


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


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

polls/urls.py ¶

```python
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"),
]
```

Escribir tu primera aplicación Django

Cuando alguien solicita una página de tu sitio web (por ejemplo, “/polls/34/”), Django cargará el **`mysite.urls`**módulo Python porque está indicado por la **`ROOT_URLCONF`**configuración. Busca la variable nombrada **`urlpatterns`** y recorre 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 da como resultado una llamada a la **`detail()`** vista de la siguiente manera:

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

La **`question_id=34`**parte 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 **`question_id`**parte de la cadena define el nombre que se usará para identificar el patrón coincidente, y la **`int`**parte es un convertidor que determina qué patrones deben coincidir con esta parte de la ruta de la URL. Los dos puntos (**`:`**) separan el convertidor y el nombre del patrón.

Escribir tu primera aplicación Django

Escribe vistas que realmente hagan algo ¶

Cada vista es responsable de hacer una de dos cosas: devolver un `HttpResponse` objeto 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 utilizar 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 Python que desee.

Aquí hay un intento de una nueva `index()` vista, que muestra las últimas 5 preguntas de la encuesta en el sistema, separadas por comas, según la fecha de publicación:

polls/views.py ¶

```python
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)
```

Escribir tu primera aplicación Django

Sin embargo, hay un problema: el diseño de la página está codificado en la vista. Si quieres cambiar el aspecto de la página, tendrás que editar este código Python. Por lo tanto, usemos el sistema de plantillas de Django para separar el diseño de Python creando una plantilla que la vista pueda usar.

Primero, crea un directorio llamado **`templates`**in your **`polls`**directory. Django buscará plantillas allí.

La configuración de su proyecto **`TEMPLATES`**describe cómo Django cargará y renderizará las plantillas. El archivo de configuración predeterminado configura un **`DjangoTemplates`** backend cuya **`APP_DIRS`**opción está establecida en **`True`**. Por convención, **`DjangoTemplates`**busca un subdirectorio “templates” en cada uno de los **`INSTALLED_APPS`**.

Dentro del **`templates`**directorio que acabas de crear, crea otro directorio llamado **`polls`**, y dentro de él crea un archivo llamado **`index.html`**. En otras palabras, tu plantilla debería estar en **`polls/templates/polls/index.html`**. Debido a cómo **`app_directories`** funciona el cargador de plantillas como se describió anteriormente, puedes hacer referencia a esta plantilla dentro de Django como **`polls/index.html`**.

Escribir tu primera aplicación Django

1 Coloque el siguiente código en esa plantilla:

2 Ahora actualicemos nuestra **`index`**vista **`polls/views.py`**para usar la 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 %}
```

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))
```

Escribir tu primera aplicación Django

Un atajo: `render()`

Es un modismo muy común cargar una plantilla, completar un contexto y devolver un `HttpResponse`objeto con el resultado de la plantilla generada. Django ofrece un atajo. Aquí se muestra la `index()` vista completa, reescrita:

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

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)
```

Escribir tu primera aplicación Django

Un atajo: `render()`

Es un modismo muy común cargar una plantilla, completar un contexto y devolver un `HttpResponse`objeto con el resultado de la plantilla generada. Django ofrece un atajo. Aquí se muestra la `index()` vista completa, reescrita:

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

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)
```

Escribir tu primera aplicación Django

1 Ahora, analicemos la vista de detalles de la pregunta, la página que muestra el texto de la pregunta de una encuesta determinada. Esta es la vista:

2 El nuevo concepto aquí: la vista genera la `Http404`excepción si no existe una pregunta con el ID solicitado.

3 Agrega

polls/views.py ¶

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

from .models import Question


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

polls/templates/polls/detail.html

```
{{ question }}
```

Escribir tu primera aplicación Django

Un atajo: `get_object_or_404()`

1 Es un modismo muy común que se utiliza `get()` y se genera `Http404`si el objeto no existe. Django ofrece un atajo. Aquí está la `detail()` vista, reescrita:

2 La `get_object_or_404()` función toma un modelo de Django como primer argumento y una cantidad arbitraria de argumentos de palabras clave, que pasa a la `get()` función del administrador del modelo. Se activa `Http404`si el objeto no existe.

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

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})
```

El sistema de plantillas (Templates)

El sistema de plantillas utiliza la sintaxis de búsqueda por puntos para acceder a los atributos de las variables. En el ejemplo de , primero Django realiza una búsqueda en el diccionario del objeto . Si no lo logra, intenta una búsqueda de atributos, que funciona en este caso. Si la búsqueda de atributos hubiera fallado, habría intentado una búsqueda en el índice de la lista.`{{ question.question_text }}question`

La llamada al método ocurre en el bucle: se interpreta como el código Python , que devuelve un iterable de objetos y es adecuado para su uso en la etiqueta.`{% for %}`

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>
```

El sistema de plantillas

Eliminar URL codificadas en plantillas ¶

Recuerde, cuando escribimos el enlace a una pregunta en la **`polls/index.html`** plantilla, el enlace estaba parcialmente codificado de esta manera:

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

El problema con este enfoque codificado de forma rígida y acoplado es que resulta complicado cambiar las URL en proyectos con muchas plantillas. Sin embargo, dado que definió el **`name`** argumento en las **`path()`** funciones del **`polls.urls`** módulo, puede eliminar la dependencia de rutas URL específicas definidas en sus configuraciones de URL mediante el uso de la etiqueta de plantilla: **`{% url %}`**

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

El sistema de plantillas

Espaciado de nombres de URL ¶

En proyectos reales de Django, puede haber cinco, diez, veinte aplicaciones o más. ¿Cómo diferencia Django los nombres de URL entre ellas? Por ejemplo, la **`polls`** aplicación tiene una **`detail`** vista, y también podría tenerla una aplicación del mismo proyecto que sea para un blog. ¿Cómo se hace para que Django sepa qué vista de aplicación crear para una URL cuando se usa la etiqueta de plantilla? **`{% url %}`**

La respuesta es agregar espacios de nombres a su URLconf. En el **`polls/urls.py`** archivo, continúe y agregue un **`app_name`** para configurar el espacio de nombres de la aplicación.

para señalar la vista de detalles con espacio de nombres: `polls/templates/polls/index.html`¶

polls/urls.py ¶

```
from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.index, name=
    path("<int:question_id>/",
    path("<int:question_id>/res
    path("<int:question_id>/vot
]
```

polls/templates/polls/index.html ¶

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

Escribe una forma mínima (Form)

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

- La plantilla anterior muestra un botón de opción para cada opción de pregunta. El **value**de cada botón de opción es el ID de la opción de pregunta asociada. El **name**de cada botón de opción es **"choice"**. Esto significa que, cuando alguien selecciona uno de los botones de opción y envía el formulario, se enviarán los datos POST **choice=#**donde # es el ID de la opción seleccionada. Este es el concepto básico de los formularios HTML.
- Establecemos el formulario **action**en y establecemos . El uso de (en lugar de) es muy importante, porque el acto de enviar este formulario alterará los datos del lado del servidor. Siempre que cree un formulario que altere los datos del lado del servidor, utilice . Este consejo no es específico de Django; es una buena práctica de desarrollo web en general.**{% url 'polls:vote' question.id %}method="post"method="post"method="get"method="post"**
- **forloop.counter**Indica cuántas veces la **for**etiqueta ha pasado por su bucle.
- Dado que estamos creando un formulario POST (que puede tener el efecto de modificar datos), debemos preocuparnos por las falsificaciones de solicitudes entre sitios. Afortunadamente, no tiene que preocuparse demasiado, porque Django viene con un sistema útil para protegerse contra ellas. En resumen, todos los formularios POST que estén dirigidos a URL internas deben usar la etiqueta de plantilla.**{% csrf_token %}**

polls/templates/polls/detail.html ¶

```
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
<fieldset>
    <legend><h1>{{ question.question_text }}</h1></legend>
    {% if error_message %}<p><strong>{{ error_message }}</strong></p>{%
endif %}
    {% 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 %}
</fieldset>
<input type="submit" value="Vote">
</form>
```

Escribe una forma mínima (Form)

creamos una URLconf para la aplicación de encuestas que incluye esta línea:

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

- **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**Los valores siempre son cadenas.

  Tenga en cuenta que Django también permite **request.GET**acceder a datos GET de la misma manera, pero lo usamos explícitamente **request.POST**en nuestro código para garantizar que los datos solo se alteren mediante una llamada POST.

- **request.POST['choice']**Se generará un error **KeyError**si **choice**no se proporcionó en los datos POST. El código anterior verifica **KeyError**y vuelve a mostrar el formulario de preguntas con un mensaje de error si **choice**no se proporciona.

- **F("votes") + 1** indica a la base de datos que aumente el recuento de votos en 1.

- Después de incrementar el número de opciones, el código devuelve un **HttpResponseRedirect**en lugar de un . normal **HttpResponse**. **HttpResponseRedirect**toma un único argumento: la URL a la que será redirigido el usuario (vea el siguiente punto para ver cómo construimos la URL en este caso).

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

- En este ejemplo , utilizamos la **reverse()**función en el **HttpResponseRedirect**constructor. Esta función ayuda a evitar tener que codificar una URL en la función de vista. Se le proporciona 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, utilizando la URLconf que configuramos en el Tutorial 3 , esta **reverse()**llamada devolverá una cadena como

  `"/polls/3/results/"`

polls/views.py ¶

```
from django.db.models import F
from django.http import HttpResponse, 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 = F("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,)))
```

Escribe una forma mínima (Form)

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

Esta es casi exactamente la misma `detail()` vista. La única diferencia es el nombre de la plantilla. Solucionaremos esta redundancia más adelante.

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})
```

Ahora, crea una `polls/results.html` plantilla:

ve a `/polls/1/`tu navegador y vota la pregunta. Deberías ver una página de resultados que se actualiza cada vez que votas. Si envías el formulario sin haber elegido una opción, deberías ver el mensaje de error.

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>
```

Vistas genéricas: menos código es mejor

Hay vistas que representan un caso común de desarrollo web básico: obtener datos de la base de datos según un parámetro pasado en la URL, cargar una plantilla y devolver la plantilla renderizada. Como esto es tan común, **Django proporciona un atajo, llamado sistema de "vistas genéricas".**

Las vistas genéricas abstraen patrones comunes hasta el punto en que ni siquiera es necesario escribir código Python para crear una aplicación. Por ejemplo, las vistas genéricas `ListView` y `DetailView` abstraen los conceptos de "mostrar una lista de objetos" y "mostrar una página de detalles para un tipo particular de objeto", respectivamente.

Convirtamos nuestra aplicación de encuestas para que utilice el sistema de vistas genérico, de modo que podamos eliminar una parte de nuestro propio código. Tendremos que realizar algunos pasos para realizar la conversión. Haremos lo siguiente:

1. Convertir la URLconf.
2. Eliminar algunas de las vistas antiguas e innecesarias.
3. Introducir nuevas vistas basadas en las vistas genéricas de Django.

Vistas genéricas: menos código es mejor

Modificar URLconf ¶

Primero, abra la `polls/urls.py` URLconf y cámbiela de la siguiente manera:

Tenga en cuenta que el nombre del patrón coincidente en las cadenas de ruta del segundo y tercer patrón ha cambiado de `<question_id>` a `<pk>`. Esto es necesario porque usaremos la `DetailView` vista genérica para reemplazar nuestras vistas `detail()` y `results()`, y espera que se llame al valor de la clave principal capturado de la URL `"pk"`.

polls/urls.py ¶

```python
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"),
]
```

Vistas genéricas

Modificar puntos de vista ¶

A continuación, eliminaremos nuestras vistas antiguas **`index`**, **`detail`** y **`results`** y utilizaremos las vistas genéricas de Django en su lugar. Para ello, abra el **`polls/views.py`** archivo y modifíquelo de la siguiente manera

polls/views.py ¶

```python
from django.db.models import F
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.
    ...
```

Vistas genéricas: menos código es mejor

Each generic view needs to know what model it will be acting upon. This is provided using either the **`model`** attribute (in this example, **`model = Question`** for **`DetailView`** and **`ResultsView`**) or by defining the **`get_queryset()`** method (as shown in **`IndexView`**).

By default, the **`DetailView`** generic view uses a template called **`<app name>/<model name>_detail.html`**. In our case, it would use the template **`"polls/question_detail.html"`**. The **`template_name`** attribute is used to tell Django to use a specific template name instead of the autogenerated default template name. We also specify the **`template_name`** for the **`results`** list view – this ensures that the results view and the detail view have a different appearance when rendered, even though they're both a **`DetailView`** behind the scenes.

Similarly, the **`ListView`** generic view uses a default template called **`<app name>/<model name>_list.html`**; we use **`template_name`** to tell **`ListView`** to use our existing **`"polls/index.html"`** template.

In previous parts of the tutorial, the templates have been provided with a context that contains the **`question`** and **`latest_question_list`** context variables. For **`DetailView`** the **`question`** variable is provided automatically – since we're using a Django model (**`Question`**), Django is able to determine an appropriate name for the context variable. However, for ListView, the automatically generated context variable is **`question_list`**. To override this we provide the **`context_object_name`** attribute, specifying that we want to use **`latest_question_list`** instead. As an alternative approach, you could change your templates to match the new default context variables – but it's a lot easier to tell Django to use the variable you want.