EBook Gratis

APRENDIZAJE Django

Free unaffiliated eBook created from **Stack Overflow contributors.**

#django

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django

It is an unofficial and free Django ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Django.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Django

Observaciones

Django se anuncia a sí mismo como "el marco web para perfeccionistas con fechas límite" y "Django facilita la creación de mejores aplicaciones web más rápidamente y con menos código". Puede verse como una arquitectura MVC. En su núcleo tiene:

- Un servidor web ligero y autónomo para desarrollo y pruebas.
- un sistema de validación y serialización de formularios que se puede traducir entre formularios HTML y valores adecuados para el almacenamiento en la base de datos
- Un sistema de plantillas que utiliza el concepto de herencia prestada de la programación orientada a objetos.
- un marco de almacenamiento en caché que puede usar cualquiera de los varios métodos de caché compatibles con las clases de middleware que pueden intervenir en varias etapas del procesamiento de solicitudes y llevar a cabo funciones personalizadas
- un sistema de despacho interno que permite que los componentes de una aplicación se comuniquen entre sí mediante señales predefinidas
- un sistema de internacionalización, que incluye traducciones de los componentes propios de Django a una variedad de idiomas
- un sistema de serialización que puede producir y leer representaciones XML y / o JSON de instancias del modelo Django
- Un sistema para ampliar las capacidades del motor de plantillas.
- una interfaz para el marco de prueba de unidad incorporado de Python

Versiones

Versión	Fecha de lanzamiento
1.11	2017-04-04
1.10	2016-08-01
1.9	2015-12-01
1.8	2015-04-01
1.7	2014-09-02
1.6	2013-11-06
1.5	2013-02-26
1.4	2012-03-23
1.3	2011-03-23

Versión	Fecha de lanzamiento
1.2	2010-05-17
1.1	2009-07-29
1.0	2008-09-03

Examples

Comenzando un proyecto

Django es un framework de desarrollo web basado en Python. Django **1.11** (la última versión estable) requiere la instalación de Python **2.7** , **3.4** , **3.5** o **3.6** . Suponiendo que `pip` está disponible, la instalación es tan simple como ejecutar el siguiente comando. Tenga en cuenta que omitir la versión como se muestra a continuación instalará la última versión de django:

```
$ pip install django
```

Para instalar una versión específica de django, supongamos que la versión es django **1.10.5** , ejecute el siguiente comando:

```
$ pip install django==1.10.5
```

Las aplicaciones web creadas con Django deben residir dentro de un proyecto de Django. Puede usar el comando `django-admin` para iniciar un nuevo proyecto en el directorio actual:

```
$ django-admin startproject myproject
```

donde `myproject` es un nombre que identifica de forma única el proyecto y puede constar de **números** , **letras** y **guiones bajos** .

Esto creará la siguiente estructura de proyecto:

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

Para ejecutar la aplicación, inicie el servidor de desarrollo.

```
$ cd myproject
$ python manage.py runserver
```

Ahora que el servidor está funcionando, visite `http://127.0.0.1:8000/` con su navegador web.

Verás la siguiente página:

De forma predeterminada, el comando `runserver` inicia el servidor de desarrollo en la IP interna en el puerto `8000` . Este servidor se reiniciará automáticamente a medida que realice cambios en su código. Pero en caso de que agregue nuevos archivos, deberá reiniciar manualmente el servidor.

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

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

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

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

Tenga en cuenta que `runserver` es solo para compilaciones de depuración y pruebas locales. Los programas de servidor especializados (como Apache) siempre deben usarse en producción.

Añadiendo una aplicación Django

Un proyecto de Django usualmente contiene múltiples `apps` . Esta es simplemente una forma de estructurar su proyecto en módulos más pequeños y mantenibles. Para crear una aplicación, vaya a su carpeta de proyecto (donde `manage.py` es), y ejecute el comando `startapp` (cambie *myapp* a lo que quiera):

```
python manage.py startapp myapp
```

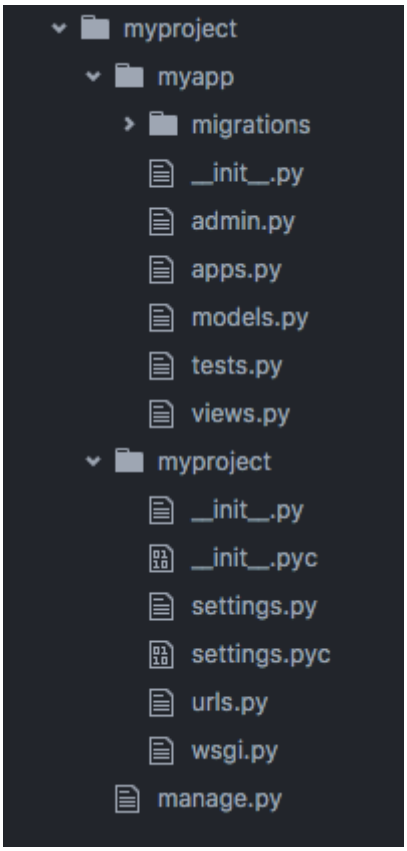
Esto generará la carpeta *myapp* y algunos archivos necesarios para usted, como `models.py` y `views.py` .

Para que Django sea consciente de *myapp* , agréguelo a su `settings.py` :

```
# myproject/settings.py

# Application definition
INSTALLED_APPS = [
    ...
    'myapp',
]
```

La estructura de carpetas de un proyecto de Django se puede cambiar para que se ajuste a sus preferencias. A veces, la carpeta del proyecto cambia de nombre a `/src` para evitar repetir los nombres de las carpetas. Una estructura de carpetas típica se ve así:



Conceptos Django

django-admin es una herramienta de línea de comandos que se envía con Django. Viene con varios comandos útiles para comenzar y administrar un proyecto Django. El comando es el mismo que `./manage.py` , con la diferencia de que no es necesario que esté en el directorio del proyecto. La variable de entorno `DJANGO_SETTINGS_MODULE` debe configurarse.

Un **proyecto de Django** es un código base de Python que contiene un archivo de configuración de Django. El administrador de Django puede crear un proyecto mediante el comando `django-admin startproject NAME` . El proyecto normalmente tiene un archivo llamado `manage.py` en el nivel superior y un archivo URL raíz llamado `urls.py` `manage.py` es una versión específica del proyecto de `django-admin` , y le permite ejecutar comandos de administración en ese proyecto. Por ejemplo, para ejecutar su proyecto localmente, use `python manage.py runserver` . Un proyecto se compone de aplicaciones Django.

Una **aplicación de Django** es un paquete de Python que contiene un archivo de modelos (`models.py` de forma predeterminada) y otros archivos, como urls y vistas específicas de la aplicación. Se puede crear una aplicación a través del comando `django-admin startapp NAME` (este comando debe ejecutarse desde el directorio de su proyecto). Para que una aplicación sea parte de un proyecto, debe incluirse en la lista de `INSTALLED_APPS` en `settings.py` . Si usó la configuración estándar, Django viene con varias aplicaciones de sus propias aplicaciones preinstaladas que manejarán cosas como la autenticación para usted. Las aplicaciones se pueden utilizar en

múltiples proyectos de Django.

El **ORM de Django** recopila todos los modelos de base de datos definidos en `models.py` y crea tablas de base de datos basadas en esas clases de modelos. Para hacer esto, primero, configure su base de datos modificando la configuración de `DATABASES` en `settings.py` . Luego, una vez que haya definido sus modelos de base de datos , ejecute `python manage.py makemigrations` seguido de `python manage.py migrate` para crear o actualizar el esquema de su base de datos según sus modelos.

Un ejemplo completo de hola mundo.

Paso 1 Si ya tienes Django instalado, puedes omitir este paso.

```
pip install Django
```

Paso 2 Crear un nuevo proyecto

```
django-admin startproject hello
```

Eso creará una carpeta llamada `hello` que contendrá los siguientes archivos:

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Paso 3 Dentro del módulo de `hello` (la carpeta que contiene el `__init.py__`) cree un archivo llamado `views.py` :

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── views.py  <- here
│   └── wsgi.py
└── manage.py
```

y poner en el siguiente contenido:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('Hello, World')
```

Esto se llama una función de vista.

Paso 4 Edita `hello/urls.py` como sigue:

```
from django.conf.urls import url
from django.contrib import admin
from hello import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.hello)
]
```

que enlaza la función de vista `hello()` a una URL.

Paso 5 Inicia el servidor.

```
python manage.py runserver
```

Paso 6

Vaya a `http://localhost:8000/` en un navegador y verá:

> Hola Mundo

Ambiente virtual

Aunque no es estrictamente necesario, se recomienda iniciar su proyecto en un "entorno virtual". Un entorno virtual es un **contenedor** (un directorio) que contiene una versión específica de Python y un conjunto de módulos (dependencias), y que no interfiere con el Python nativo del sistema operativo u otros proyectos en la misma computadora.

Al configurar un entorno virtual diferente para cada proyecto en el que trabaja, varios proyectos de Django pueden ejecutarse en diferentes versiones de Python y pueden mantener sus propios conjuntos de dependencias, sin riesgo de conflicto.

Python 3.3+

Python 3.3+ ya incluye un módulo `venv` estándar, al que normalmente puedes llamar como `pyvenv` . En entornos donde el comando `pyvenv` no está disponible, puede acceder a la misma funcionalidad invocando directamente el módulo como `python3 -m venv` .

Para crear el entorno virtual:

```
$ pyvenv <env-folder>
# Or, if pyvenv is not available
$ python3 -m venv <env-folder>
```

Python 2

Si usa Python 2, primero puede instalarlo como un módulo separado de pip:

```
$ pip install virtualenv
```

Y luego cree el entorno usando el comando `virtualenv` lugar:

```
$ virtualenv <env-folder>
```

Activar (cualquier versión)

El entorno virtual ya está configurado. Para poder utilizarlo, debe estar *activado* en el terminal que desea utilizar.

Para 'activar' el entorno virtual (cualquier versión de Python)

Linux como:

```
$ source <env-folder>/bin/activate
```

Windows como:

```
<env-folder>\Scripts\activate.bat
```

Esto cambia su indicador para indicar que el entorno virtual está activo. `(<env-folder>) $`

De ahora en adelante, todo lo que se instale usando `pip` se instalará en su carpeta de env virtual, no en todo el sistema.

Para salir del entorno virtual utilice `deactivate` :

```
(<env-folder>) $ deactivate
```

Alternativamente: use virtualenvwrapper

También puede considerar el uso de virtualenvwrapper, que hace que la creación y activación de virtualenv sea muy útil, así como la separación de su código:

```
# Create a virtualenv
mkvirtualenv my_virtualenv

# Activate a virtualenv
workon my_virtualenv

# Deactivate the current virtualenv
deactivate
```

Alternativamente: use pyenv + pyenv-

virtualenv

En entornos en los que necesita manejar varias versiones de Python, puede beneficiarse de virtualenv junto con pyenv-virtualenv:

```
# Create a virtualenv for specific Python version
pyenv virtualenv 2.7.10 my-virtual-env-2.7.10

# Create a vritualenv for active python verion
pyenv virtualenv venv34

# Activate, deactivate virtualenv
pyenv activate <name>
pyenv deactivate
```

Cuando se utiliza virtualenvs, a menudo es útil configurar `PYTHONPATH` y `DJANGO_SETTINGS_MODULE` en el script `postactivate` .

```
#!/bin/sh
# This hook is sourced after this virtualenv is activated

# Set PYTHONPATH to isolate the virtualenv so that only modules installed
# in the virtualenv are available
export PYTHONPATH="/home/me/path/to/your/project_root:$VIRTUAL_ENV/lib/python3.4"

# Set DJANGO_SETTINGS_MODULE if you don't use the default `myproject.settings`
# or if you use `django-admin` rather than `manage.py`
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Establece la ruta de tu proyecto

A menudo también es útil configurar la ruta de su proyecto dentro de un archivo `.project` especial ubicado en su `<env-folder>` . Al hacer esto, cada vez que active su entorno virtual, cambiará el directorio activo a la ruta especificada.

Cree un nuevo archivo llamado `<env-folder>/.project` . El contenido del archivo SOLO debe ser la ruta del directorio del proyecto.

```
/path/to/project/directory
```

Ahora, inicie su entorno virtual (ya sea utilizando `source <env-folder>/bin/activate` `workon my_virtualenv` o `workon my_virtualenv`) y su terminal cambiará los directorios a `/path/to/project/directory` .

Un solo archivo Hello World Ejemplo

Este ejemplo muestra una forma mínima de crear una página Hello World en Django. Esto le ayudará a darse cuenta de que el comando de `django-admin startproject example` crea

básicamente un montón de carpetas y archivos y que no necesariamente necesita esa estructura para ejecutar su proyecto.

1. Crea un archivo llamado `file.py`
2. Copia y pega el siguiente código en ese archivo.

```
import sys

from django.conf import settings

settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.http import HttpResponse

# Your code goes below this line.

def index(request):
    return HttpResponse('Hello, World!')

urlpatterns = [
    url(r'^$', index),
]

# Your code goes above this line

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

3. Vaya a la terminal y ejecute el archivo con este comando `python file.py runserver` .
4. Abra su navegador y vaya a 127.0.0.1:8000 .

Proyecto de implementación amigable con soporte Docker.

La plantilla de proyecto Django predeterminada está bien, pero una vez que implementas tu código y, por ejemplo, los desarrolladores ponen sus manos en el proyecto, las cosas se complican. Lo que puede hacer es separar su código fuente del resto que se requiere para estar en su repositorio.

> Puede encontrar una plantilla de proyecto Django utilizable en GitHub .

Estructura del proyecto

```
PROJECT_ROOT
├── devel.dockerfile
├── docker-compose.yml
├── nginx
│   └── project_name.conf
├── README.md
├── setup.py
└── src
    ├── manage.py
    └── project_name
        ├── __init__.py
        └── service
            ├── __init__.py
            ├── settings
            │   ├── common.py
            │   ├── development.py
            │   ├── __init__.py
            │   └── staging.py
            ├── urls.py
            └── wsgi.py
```

Me gusta mantener el directorio de `service` denominado `service` para cada proyecto gracias a que puedo usar el mismo `Dockerfile` en todos mis proyectos. La división de requisitos y configuraciones ya está bien documentada aquí:
Usando múltiples archivos de requerimientos
Usando múltiples configuraciones

Dockerfile

Con el supuesto de que solo los desarrolladores hacen uso de Docker (no todos los desarrolladores de operaciones confían en ello en estos días). Esto podría ser un entorno de desarrollo `devel.dockerfile` :

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1

RUN mkdir /run/service
ADD . /run/service
WORKDIR /run/service

RUN pip install -U pip
RUN pip install -I -e .[develop] --process-dependency-links

WORKDIR /run/service/src
ENTRYPOINT ["python", "manage.py"]
CMD ["runserver", "0.0.0.0:8000"]
```

Agregar solo los requisitos aprovechará la memoria caché de Docker mientras se construye, solo necesita reconstruir el cambio de requisitos.

Componer

Docker compose es muy útil, especialmente cuando tiene múltiples servicios para ejecutar localmente. `docker-compose.yml` :

```
version: '2'
services:
  web:
    build:
      context: .
      dockerfile: devel.dockerfile
    volumes:
      - "./src/{{ project_name }}:/run/service/src/{{ project_name }}"
      - "./media:/run/service/media"
    ports:
      - "8000:8000"
    depends_on:
      - db
  db:
    image: mysql:5.6
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE={{ project_name }}
  nginx:
    image: nginx
    ports:
      - "80:80"
    volumes:
      - "./nginx:/etc/nginx/conf.d"
      - "./media:/var/media"
    depends_on:
      - web
```

Nginx

Su entorno de desarrollo debe ser lo más cercano posible al entorno prod, por lo que me gusta usar Nginx desde el principio. Aquí hay un ejemplo de archivo de configuración nginx:

```
server {
    listen   80;
    client_max_body_size 4G;
    keepalive_timeout 5;

    location /media/ {
        autoindex on;
        alias /var/media/;
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
```

```
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Ssl on;
        proxy_connect_timeout 600;
        proxy_read_timeout 600;
        proxy_pass http://web:8000/;
    }
}
```

Uso

```
$ cd PROJECT_ROOT
$ docker-compose build web  # build the image - first-time and after requirements change
$ docker-compose up  # to run the project
$ docker-compose run --rm --service-ports --no-deps  # to run the project - and be able to use
PDB
$ docker-compose run --rm --no-deps <management_command>  # to use other than runserver
commands, like makemigrations
$ docker exec -ti web bash  # For accessing django container shell, using it you will be
inside /run/service directory, where you can run ./manage shell, or other stuff
$ docker-compose start  # Starting docker containers
$ docker-compose stop  # Stopping docker containers
```

Lea Empezando con Django en línea: https://riptutorial.com/es/django/topic/200/empezando-con-django

Capítulo 2: ¿Cómo usar Django con Cookiecutter?

Examples

Instalando y configurando el proyecto django usando Cookiecutter

Los siguientes son los requisitos previos para instalar Cookiecutter:

- pipa
- virtualenv
- PostgreSQL

Crea un virtualenv para tu proyecto y actívalo:

```
$ mkvirtualenv <virtualenv name>
$ workon <virtualenv name>
```

Ahora instale Cookiecutter usando:

```
$ pip install cookiecutter
```

Cambie los directorios a la carpeta donde desea que viva su proyecto. Ahora ejecuta el siguiente comando para generar un proyecto django:

```
$ cookiecutter https://github.com/pydanny/cookiecutter-django.git
```

Este comando ejecuta a cookiecutter con el repositorio de cookiecutter-django, solicitándonos que ingresemos detalles específicos del proyecto. Presione "enter" sin escribir nada para usar los valores predeterminados, que se muestran entre [corchetes] después de la pregunta.

```
project_name [project_name]: example_project
repo_name [example_project]:
author_name [Your Name]: Atul Mishra
email [Your email]: abc@gmail.com
description [A short description of the project.]: Demo Project
domain_name [example.com]: example.com
version [0.1.0]: 0.1.0
timezone [UTC]: UTC
now [2016/03/08]: 2016/03/08
year [2016]: 2016
use_whitenoise [y]: y
use_celery [n]: n
use_mailhog [n]: n
use_sentry [n]: n
use_newrelic [n]: n
use_opbeat [n]: n
windows [n]: n
```

```
use_python2 [n]: n
```

Más detalles sobre las opciones de generación de proyectos se pueden encontrar en la documentación oficial. El proyecto ya está configurado.

Lea ¿Cómo usar Django con Cookiecutter? en línea: https://riptutorial.com/es/django/topic/5385/-como-usar-django-con-cookiecutter-

Capítulo 3: Administración

Examples

Lista de cambios

Digamos que tienes una aplicación de `myblog` simple con el siguiente modelo:

```
from django.conf import settings
from django.utils import timezone

class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70, unique=True)
    author = models.ForeignKey(settings.AUTH_USER_MODEL, models.PROTECT)
    date_published = models.DateTimeField(default=timezone.now)
    is_draft = models.BooleanField(default=True)
    content = models.TextField()
```

La "lista de cambios" de Django Admin es la página que lista todos los objetos de un modelo dado.

```
from django.contrib import admin
from myblog.models import Article

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    pass
```

Por defecto, utilizará el `__str__()` (o `__unicode__()` si está en python2) de su modelo para mostrar el "nombre" del objeto. Esto significa que si no lo ha anulado, verá una lista de artículos, todos llamados "Objeto de artículo". Para cambiar este comportamiento, puede configurar el `__str__()` :

```
class Article(models.Model):
    def __str__(self):
        return self.title
```

Ahora, todos sus artículos deben tener un nombre diferente y más explícito que "Objeto de artículo".

Sin embargo, es posible que desee mostrar otros datos en esta lista. Para esto, usa `list_display` :

```
@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['__str__', 'author', 'date_published', 'is_draft']
```

`list_display` no está limitado a los campos y propiedades del modelo. También puede ser un método de su `ModelAdmin` :

```
from django.forms.utils import flatatt
from django.urls import reverse
from django.utils.html import format_html

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'author_link', 'date_published', 'is_draft']

    def author_link(self, obj):
        author = obj.author
        opts = author._meta
        route = '{}_{}_change'.format(opts.app_label, opts.model_name)
        author_edit_url = reverse(route, args=[author.pk])
        return format_html(
            '<a{}>{}</a>', flatatt({'href': author_edit_url}), author.first_name)

    # Set the column name in the change list
    author_link.short_description = "Author"
    # Set the field to use when ordering using this column
    author_link.admin_order_field = 'author__firstname'
```

Estilos CSS adicionales y scripts JS para la página de administración

Supongamos que tiene un modelo de `Customer` simple:

```
class Customer(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    is_premium = models.BooleanField(default=False)
```

Se registra en la administración de Django y añadir campo de búsqueda por `first_name` y `last_name` :

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']
```

Después de hacer esto, los campos de búsqueda aparecen en la página de la lista de administradores con el marcador de posición predeterminado: " *palabra clave* ". ¿Pero qué sucede si desea cambiar ese marcador de posición a " *Buscar por nombre* "?

Puede hacerlo pasando un archivo Javascript personalizado a los `Media` administración:

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']

    class Media:
        #this path may be any you want,
        #just put it in your static folder
        js = ('js/admin/placeholder.js', )
```

Puede usar la barra de herramientas de depuración del navegador para encontrar qué ID o clase Django estableció en esta barra de búsqueda y luego escribir su código js:

```
$(function () {
   $('#searchbar').attr('placeholder', 'Search by name')
})
```

También la clase `Media` permite agregar archivos css con objeto de diccionario:

```
class Media:
    css = {
        'all': ('css/admin/styles.css',)
         }
```

Por ejemplo, necesitamos mostrar cada elemento de la columna `first_name` en un color específico.

Por defecto, Django crea una columna en la tabla para cada elemento en `list_display` , todas las etiquetas `<td>` tendrán una clase css como el `field-'list_display_name'` , en nuestro caso será `field_first_name`

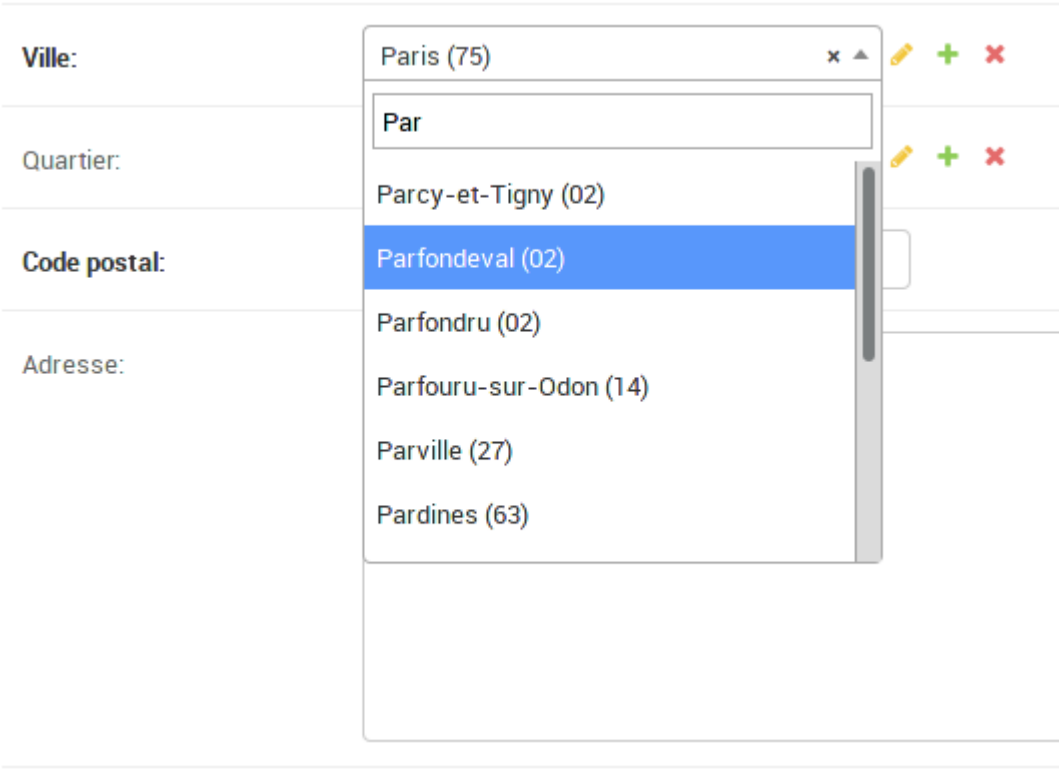
```
.field_first_name {
     background-color: #e6f2ff;
 }
```

Si desea personalizar otro comportamiento agregando JS o algunos estilos css, siempre puede verificar los ID y las clases de elementos en la herramienta de depuración del navegador.

Manejo de claves foráneas que hacen referencia a tablas grandes.

De forma predeterminada, Django presenta los campos de `ForeignKey` como una entrada de `<select>` . Esto puede hacer que las páginas se **carguen muy lentamente** si tiene miles o decenas de miles de entradas en la tabla a la que se hace referencia. E incluso si tiene solo cientos de entradas, es bastante incómodo buscar una entrada en particular entre todas.

Un módulo externo muy útil para esto es django-autocomplete-light (DAL). Esto permite utilizar los campos de autocompletar en lugar de los campos de `<select>` .



vistas.py

```
from dal import autocomplete

class CityAutocomp(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        qs = City.objects.all()
        if self.q:
            qs = qs.filter(name__istartswith=self.q)
        return qs
```

urls.py

```
urlpatterns = [
    url(r'^city-autocomp/$', CityAutocomp.as_view(), name='city-autocomp'),
]
```

forms.py

```
from dal import autocomplete

class PlaceForm(forms.ModelForm):
    city = forms.ModelChoiceField(
        queryset=City.objects.all(),
        widget=autocomplete.ModelSelect2(url='city-autocomp')
    )

    class Meta:
        model = Place
```

```
        fields = ['__all__']
```

admin.py

```
@admin.register(Place)
class PlaceAdmin(admin.ModelAdmin):
    form = PlaceForm
```

Lea Administración en línea: https://riptutorial.com/es/django/topic/1219/administracion

Capítulo 4: Agregaciones de modelos

Introducción

Las agregaciones son métodos que permiten la ejecución de operaciones en (individuales y / o grupos de) filas de objetos derivados de un Modelo.

Examples

Promedio, mínimo, máximo, suma de Queryset

```
class Product(models.Model):
    name = models.CharField(max_length=20)
    price = models.FloatField()
```

Para obtener el precio promedio de todos los productos:

```
>>> from django.db.models import Avg, Max, Min, Sum
>>> Product.objects.all().aggregate(Avg('price'))
# {'price__avg': 124.0}
```

Para obtener el precio mínimo de todos los productos:

```
>>> Product.objects.all().aggregate(Min('price'))
# {'price__min': 9}
```

Para obtener el precio máximo de todos los productos:

```
>>> Product.objects.all().aggregate(Max('price'))
# {'price__max':599 }
```

Para obtener SUMA de precios de todos los productos:

```
>>> Product.objects.all().aggregate(Sum('price'))
# {'price__sum':92456 }
```

Cuenta el número de relaciones exteriores.

```
class Category(models.Model):
    name = models.CharField(max_length=20)


class Product(models.Model):
    name = models.CharField(max_length=64)
    category = models.ForeignKey(Category, on_delete=models.PROTECT)
```

Para obtener el número de productos para cada categoría:

```
>>> categories = Category.objects.annotate(Count('product'))
```

Esto agrega el atributo `<field_name>__count` a cada instancia devuelta:

```
>>> categories.values_list('name', 'product__count')
[('Clothing', 42), ('Footwear', 12), ...]
```

Puede proporcionar un nombre personalizado para su atributo utilizando un argumento de palabra clave:

```
>>> categories = Category.objects.annotate(num_products=Count('product'))
```

Puede utilizar el campo anotado en querysets:

```
>>> categories.order_by('num_products')
[<Category: Footwear>, <Category: Clothing>]

>>> categories.filter(num_products__gt=20)
[<Category: Clothing>]
```

GROUB BY ... COUNT / SUM Django ORM equivalente

Podemos realizar un `GROUP BY ... COUNT` o un `GROUP BY ... SUM` consultas SQL equivalentes en ORM de Django, con el uso de `annotate()` , `values()` , `order_by()` y los `django.db.models` 's `Count` y `Sum` respetuosamente los métodos:

Deja que nuestro modelo sea:

```
   class Books(models.Model):
       title  = models.CharField()
       author = models.CharField()
       price = models.FloatField()
```

`GROUP BY ... COUNT` :

- Supongamos que queremos contar cuántos objetos de libros por autor distinto existen en nuestra tabla de `Books` :

  ```
  result = Books.objects.values('author')
                        .order_by('author')
                        .annotate(count=Count('author'))
  ```

- Ahora el `result` contiene un queryset con dos columnas: `author` y `count` :

  ```
    author     | count
  -------------|-------
   OneAuthor   |   5
  OtherAuthor  |   2
     ...       |  ...
  ```

`GROUB BY ... SUM` :

- Supongamos que queremos sumar el precio de todos los libros por autor distinto que existe en nuestra tabla de `Books` :

```
result = Books.objects.values('author')
                      .order_by('author')
                      .annotate(total_price=Sum('price'))
```

- Ahora el `result` contiene un queryset con dos columnas: `author` y `total_price` :

```
  author     | total_price
-------------|-------------
 OneAuthor   |    100.35
OtherAuthor  |     50.00
    ...      |      ...
```

Lea Agregaciones de modelos en línea: https://riptutorial.com/es/django/topic/3775/agregaciones-de-modelos

Capítulo 5: Ajustes

Examples

Configuración de la zona horaria

Puedes establecer la zona horaria que usará Django en el archivo `settings.py` . Ejemplos:

```
TIME_ZONE = 'UTC'  # use this, whenever possible
TIME_ZONE = 'Europe/Berlin'
TIME_ZONE = 'Etc/GMT+1'
```

Aquí está la lista de zonas horarias válidas

*Cuando se ejecuta en un entorno **Windows** , debe configurarse de la misma manera que la **zona horaria del sistema** .*

Si no quieres que Django use tiempos de datos conscientes de la zona horaria:

```
USE_TZ = False
```

Las mejores prácticas de Django requieren el uso de `UTC` para almacenar información en la base de datos:

> Incluso si su sitio web está disponible en una sola zona horaria, sigue siendo una buena práctica almacenar datos en UTC en su base de datos. La razón principal es el horario de verano (DST). Muchos países tienen un sistema de horario de verano, donde los relojes se mueven hacia adelante en primavera y hacia atrás en otoño. Si está trabajando en la hora local, es probable que encuentre errores dos veces al año, cuando ocurren las transiciones.

https://docs.djangoproject.com/en/stable/topics/i18n/timezones/

Accediendo a las configuraciones

Una vez que tenga todas sus configuraciones, querrá usarlas en su código. Para hacerlo, agregue la siguiente importación a su archivo:

```
from django.conf import settings
```

Luego puede acceder a su configuración como atributos del módulo de `settings` , por ejemplo:

```
if not settings.DEBUG:
    email_user(user, message)
```

Usando BASE_DIR para asegurar la portabilidad de la aplicación

Es una mala idea para las rutas de código duro en su aplicación. Siempre se deben usar direcciones URL relativas para que su código pueda funcionar sin problemas en diferentes máquinas. La mejor manera de configurar esto es definir una variable como esta

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

Luego use esta variable `BASE_DIR` para definir todas sus otras configuraciones.

```
TEMPLATE_PATH = os.path.join(BASE_DIR, "templates")
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),

]
```

Y así. Esto garantiza que pueda transferir su código a diferentes máquinas sin ninguna preocupación.

Sin embargo, `os.path` es un poco detallado. Por ejemplo, si su módulo de configuración es `project.settings.dev` , tendrá que escribir:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.dirname(__file__)))
```

Una alternativa es utilizar el módulo `unipath` (que puede instalar con `pip install unipath`).

```
from unipath import Path

BASE_DIR = Path(__file__).ancestor(2)  # or ancestor(3) if using a submodule

TEMPLATE_PATH = BASE_DIR.child('templates')
STATICFILES_DIRS = [
    BASE_DIR.child('static'),
]
```

Uso de variables de entorno para gestionar la configuración en servidores

El uso de variables de entorno es una forma muy utilizada para configurar la configuración de una aplicación en función de su entorno, como se indica en la aplicación The Twelve-Factor .

Como es probable que las configuraciones cambien entre los entornos de implementación, esta es una forma muy interesante de modificar la configuración sin tener que buscar en el código fuente de la aplicación, así como guardar secretos fuera de los archivos de la aplicación y el repositorio de código fuente.

En Django, la configuración principal se encuentra como `settings.py` en la carpeta de su proyecto. Como es un simple archivo de Python, puede usar el módulo `os` de Python de la biblioteca estándar para acceder al entorno (e incluso tener los valores predeterminados adecuados).

settings.py

```
import os

SECRET_KEY = os.environ.get('APP_SECRET_KEY', 'unsafe-secret-key')

DEBUG = bool(os.environ.get('DJANGO_DEBUG', True) == 'False')

ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS', '').split()

DATABASES = {
    'default': {
        'ENGINE': os.environ.get('APP_DB_ENGINE', 'django.db.backends.sqlite3'),
        'NAME': os.environ.get('DB_NAME', 'db.sqlite'),
        'USER': os.environ.get('DB_USER', ''),
        'PASSWORD': os.environ.get('DB_PASSWORD', ''),
        'HOST': os.environ.get('DB_HOST', None),
        'PORT': os.environ.get('DB_PORT', None),
        'CONN_MAX_AGE': 600,
    }
}
```

Con Django, puede cambiar la tecnología de su base de datos, de modo que puede usar sqlite3 en su máquina de desarrollo (y eso debería ser un buen valor predeterminado para comprometerse con un sistema de control de origen). Aunque esto es posible no es aconsejable:

> Los servicios de respaldo, como la base de datos de la aplicación, el sistema de colas o el caché, es un área donde la paridad dev / prod es importante. (La aplicación de los doce factores - Paridad de desarrollo / prod)

Para utilizar un parámetro DATABASE_URL para la conexión a la base de datos, consulte el ejemplo relacionado .

Usando múltiples configuraciones

El diseño de proyecto predeterminado de Django crea un solo `settings.py` . Esto suele ser útil para dividirlo así:

```
myprojectroot/
    myproject/
        __init__.py
        settings/
            __init__.py
            base.py
            dev.py
            prod.py
            tests.py
```

Esto le permite trabajar con diferentes configuraciones según esté en desarrollo, producción, pruebas o lo que sea.

Al pasar del diseño predeterminado a este diseño, la `settings.py` original.py se convierte en `settings/base.py` Cuando todos los demás submódulos "subclase" `settings/base.py` comenzando con `from .base import *` . Por ejemplo, aquí es cómo se ve la `settings/dev.py` :

```
# -*- coding: utf-8 -*-
from .base import *  # noqa

DEBUG = True
INSTALLED_APPS.extend([
    'debug_toolbar',
])
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
INTERNAL_IPS = ['192.168.0.51', '192.168.0.69']
```

Alternativa # 1

Para que `django-admin` comandos de `django-admin` funcionen correctamente, tendrá que configurar la variable de entorno `DJANGO_SETTINGS_MODULE` (que por defecto es `myproject.settings`). En desarrollo, lo establecerá en `myproject.settings.dev` . En producción, lo establecerá en `myproject.settings.prod` . Si usa un virtualenv, lo mejor es establecerlo en su script `postactivate` :

```
#!/bin/sh
export PYTHONPATH="/home/me/django_projects/myproject:$VIRTUAL_ENV/lib/python3.4"
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Si desea usar un módulo de configuración que no es señalado por `DJANGO_SETTINGS_MODULE` por una vez, puede usar la opción `--settings` de `django-admin` :

```
django-admin test --settings=myproject.settings.tests
```

Alternativa # 2

Si quieres dejar `DJANGO_SETTINGS_MODULE` en su configuración por defecto (`myproject.settings`), puede simplemente decir la `settings` módulo de la configuración que se cargue mediante la colocación de la importación en su `__init__.py` archivo.

En el ejemplo anterior, el mismo resultado se podría lograr al tener un `__init__.py` configurado para:

```
from .dev import *
```

Usando múltiples archivos de requerimientos

Cada archivo de requisitos debe coincidir con el nombre de un archivo de configuración. Lea Uso de configuraciones múltiples para más información.

Estructura

```
djangoproject
```

```
├── config
│   ├── __init__.py
│   ├── requirements
│   │   ├── base.txt
│   │   ├── dev.txt
│   │   ├── test.txt
│   │   └── prod.txt
│   └── settings
└── manage.py
```

En el archivo `base.txt` , coloque las dependencias utilizadas en todos los entornos.

```
# base.txt
Django==1.8.0
psycopg2==2.6.1
jinja2==2.8
```

Y en todos los demás archivos, incluya las dependencias base con `-r base.txt` y agregue las dependencias específicas necesarias para el entorno actual.

```
# dev.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in dev env
django-queryinspect==0.1.0
```

```
# test.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in test env
nose==1.3.7
django-nose==1.4
```

```
# prod.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in production env
django-queryinspect==0.1.0
gunicorn==19.3.0
django-storages-redux==1.3
boto==2.38.0
```

Finalmente, para instalar dependencias. Ejemplo, en dev env: `pip install -r config/requirements/dev.txt`

Ocultar datos secretos usando un archivo JSON

Al usar un VCS como Git o SVN, hay algunos datos secretos que nunca deben ser versionados (ya sea que el repositorio sea público o privado).

Entre esos datos, se encuentra la configuración `SECRET_KEY` y la contraseña de la base de datos.

Una práctica común para ocultar estas configuraciones del control de versiones es crear un archivo `secrets.json` en la raíz de su proyecto (gracias a la idea) " *Two Scoops of Django* " :

```
{
    "SECRET_KEY": "N4HE:AMk:.Ader5354DR453TH8SHTQr",
    "DB_PASSWORD": "v3ry53cr3t"
}
```

Y `.gitignore` a tu lista de ignorados (`.gitignore` para git):

```
*.py[co]
*.sw[po]
*~
/secrets.json
```

Luego agregue la siguiente función a su módulo de `settings` :

```
import json
import os
from django.core.exceptions import ImproperlyConfigured

with open(os.path.join(BASE_DIR, 'secrets.json')) as secrets_file:
    secrets = json.load(secrets_file)

def get_secret(setting, secrets=secrets):
    """Get secret setting or fail with ImproperlyConfigured"""
    try:
        return secrets[setting]
    except KeyError:
        raise ImproperlyConfigured("Set the {} setting".format(setting))
```

Luego llene la configuración de esta manera:

```
SECRET_KEY = get_secret('SECRET_KEY')
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgres',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': get_secret('DB_PASSWORD'),
    },
}
```

Créditos: Two Scoops of Django: Best Practices for Django 1.8, de Daniel Roy Greenfeld y Audrey RoyGreenfeld. Copyright 2015 Two Scoops Press (ISBN 978-0981467344)

Usando un DATABASE_URL del entorno

En sitios de PaaS como Heroku, es habitual recibir la información de la base de datos como una única variable de entorno de URL, en lugar de varios parámetros (host, puerto, usuario, contraseña ...).

Hay un módulo, `dj_database_url` , que extrae automáticamente la variable de entorno DATABASE_URL a un diccionario de Python apropiado para inyectar la configuración de la base de datos en Django.

Uso:

```
import dj_database_url

if os.environ.get('DATABASE_URL'):
    DATABASES['default'] =
        dj_database_url.config(default=os.environ['DATABASE_URL'])
```

Lea Ajustes en línea: https://riptutorial.com/es/django/topic/942/ajustes

Capítulo 6: ArrayField - un campo específico de PostgreSQL

Sintaxis

- desde django.contrib.postgres.fields importa ArrayField
- clase ArrayField (campo base, tamaño = Ninguno, ** opciones)
- FooModel.objects.filter (array_field_name__contains = [objetos, to, check])
- FooModel.objects.filter (array_field_name__contained_by = [objetos, a, verificar])

Observaciones

Tenga en cuenta que aunque el parámetro de `size` se pasa a PostgreSQL, PostgreSQL no lo aplicará.

Cuando use `ArrayField` s, debe tener en cuenta esta palabra de advertencia de la documentación de matrices de Postgresql .

> Consejo: Las matrices no son conjuntos; la búsqueda de elementos de una matriz específica puede ser un signo de mal diseño de la base de datos. Considere usar una tabla separada con una fila para cada elemento que sería un elemento de matriz. Esto será más fácil de buscar y es probable que se escale mejor para una gran cantidad de elementos.

Examples

Un ArrayField básico

Para crear un ArrayField de PostgreSQL, deberíamos darle a ArrayField el tipo de datos que queremos que almacene como un primer campo como campo. Ya que estaremos almacenando calificaciones de libros, usaremos `FloatField` .

```
from django.db import models, FloatField
from django.contrib.postgres.fields import ArrayField

class Book(models.Model):
    ratings = ArrayField(FloatField())
```

Especificando el tamaño máximo de un ArrayField

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class IceCream(models.Model):
    scoops = ArrayField(IntegerField()  # we'll use numbers to ID the scoops
```

```
                    , size=6)  # our parlor only lets you have 6 scoops
```

Cuando usas el parámetro size, se pasa a postgresql, que lo acepta y luego lo ignora. Por lo tanto, es muy posible agregar 7 enteros al campo de `scoops` arriba usando la consola postgresql.

Consultando la pertenencia a ArrayField con contiene

Esta consulta devuelve todos los conos con una cucharada de chocolate y una cucharada de vainilla.

```
VANILLA, CHOCOLATE, MINT, STRAWBERRY = 1, 2, 3, 4  # constants for flavors
choco_vanilla_cones = IceCream.objects.filter(scoops__contains=[CHOCOLATE, VANILLA])
```

No olvide importar el modelo `IceCream` desde su archivo `models.py` .

También tenga en cuenta que django no creará un índice para `ArrayField` s. Si va a buscarlos, necesitará un índice y tendrá que crearlo manualmente con una llamada a RunSQL en su archivo de migraciones.

Matrices de nidificación

Puede anidar `ArrayField` s pasando otro `ArrayField` como `base_field` .

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class SudokuBoard(models.Model):
    numbers = ArrayField(
        ArrayField(
            models.IntegerField(),
            size=9,
        ),
        size=9,
    )
```

Consultar a todos los modelos que contengan cualquier artículo en una lista con contenido por

Esta consulta devuelve todos los conos con una cucharada de menta o una cucharada de vainilla.

```
minty_vanilla_cones = IceCream.objects.filter(scoops__contained_by=[MINT, VANILLA])
```

Lea ArrayField - un campo específico de PostgreSQL en línea:
https://riptutorial.com/es/django/topic/1693/arrayfield---un-campo-especifico-de-postgresql

Capítulo 7: Backends de autenticación

Examples

Backend de autenticación de correo electrónico

La autenticación predeterminada de Django funciona en los campos de `username` de `username` y `password` . El backend de autenticación de correo electrónico autenticará a los usuarios según el `email` y la `password` .

```
from django.contrib.auth import get_user_model

class EmailBackend(object):
    """
    Custom Email Backend to perform authentication via email
    """
    def authenticate(self, username=None, password=None):
        user_model = get_user_model()
        try:
            user = user_model.objects.get(email=username)
            if user.check_password(password): # check valid password
                return user # return user to be authenticated
        except user_model.DoesNotExist: # no matching user exists
            return None

    def get_user(self, user_id):
        user_model = get_user_model()
        try:
            return user_model.objects.get(pk=user_id)
        except user_model.DoesNotExist:
            return None
```

Agregue este backend de autenticación a la configuración `AUTHENTICATION_BACKENDS` .

```
# settings.py
AUTHENTICATION_BACKENDS = (
    'my_app.backends.EmailBackend',
    ...
    )
```

Lea Backends de autenticación en línea: https://riptutorial.com/es/django/topic/1282/backends-de-autenticacion

Capítulo 8: Comandos de gestión

Introducción

Los comandos de administración son scripts potentes y flexibles que pueden realizar acciones en su proyecto Django o en la base de datos subyacente. ¡Además de varios comandos predeterminados, es posible escribir los tuyos!

En comparación con los scripts de Python normales, el uso del marco de comandos de administración significa que un trabajo de configuración tedioso se realiza automáticamente entre bastidores.

Observaciones

Los comandos de gestión se pueden llamar desde:

- `django-admin <command> [options]`
- `python -m django <command> [options]`
- `python manage.py <command> [options]`
- `./manage.py <command> [options]` si manage.py tiene permisos de ejecución (`chmod +x manage.py`)

Para utilizar comandos de gestión con Cron:

```
*/10 * * * * pythonuser /var/www/dev/env/bin/python /var/www/dev/manage.py <command> [options]
> /dev/null
```

Examples

Creación y ejecución de un comando de gestión

Para realizar acciones en Django utilizando la línea de comandos u otros servicios (donde no se usa el usuario / solicitud), puede usar los `management commands` .

Los módulos Django se pueden importar según sea necesario.

Para cada comando se necesita crear un archivo separado:
`myapp/management/commands/my_command.py`
(Los directorios de `management` y `commands` deben tener un archivo __init__.py vacío)

```
from django.core.management.base import BaseCommand, CommandError

# import additional classes/modules as needed
# from myapp.models import Book

class Command(BaseCommand):
    help = 'My custom django management command'
```

```
    def add_arguments(self, parser):
       parser.add_argument('book_id', nargs='+', type=int)
       parser.add_argument('author' , nargs='+', type=str)

    def handle(self, *args, **options):
       bookid = options['book_id']
       author = options['author']
       # Your code goes here

        # For example:
        # books = Book.objects.filter(author="bob")
        # for book in books:
        #    book.name = "Bob"
        #    book.save()
```

Aquí es obligatorio el nombre de clase **Comando,** que amplía **BaseCommand** o una de sus subclases.

El nombre del comando de administración es el nombre del archivo que lo contiene. Para ejecutar el comando en el ejemplo anterior, use lo siguiente en el directorio de su proyecto:

```
python manage.py my_command
```

> Tenga en cuenta que iniciar un comando puede demorar unos segundos (debido a la importación de los módulos). Entonces, en algunos casos, se recomienda crear procesos de `daemon` lugar de `management commands` de `management commands` .

Más sobre los comandos de gestión

Obtener lista de comandos existentes

Puede obtener la lista de comandos disponibles de la siguiente manera:

```
>>> python manage.py help
```

Si no entiende ningún comando o busca argumentos opcionales, puede usar el argumento **-h** como este

```
>>> python manage.py command_name -h
```

Aquí command_name será su nombre de comando deseado, esto le mostrará el texto de ayuda del comando.

```
>>> python manage.py runserver -h
>>> usage: manage.py runserver [-h] [--version] [-v {0,1,2,3}]
                           [--settings SETTINGS] [--pythonpath PYTHONPATH]
                           [--traceback] [--no-color] [--ipv6] [--nothreading]
                           [--noreload] [--nostatic] [--insecure]
                           [addrport]

Starts a lightweight Web server for development and also serves static files.
```

```
positional arguments:
  addrport              Optional port number, or ipaddr:port

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v {0,1,2,3}, --verbosity {0,1,2,3}
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on CommandError exceptions
  --no-color            Don't colorize the command output.
  --ipv6, -6            Tells Django to use an IPv6 address.
  --nothreading         Tells Django to NOT use threading.
  --noreload            Tells Django to NOT use the auto-reloader.
  --nostatic            Tells Django to NOT automatically serve static files
                        at STATIC_URL.
  --insecure            Allows serving static files even if DEBUG is False.
```

Lista de comandos disponibles

Usando django-admin en lugar de manage.py

Puede deshacerse de `manage.py` y usar el comando `django-admin` lugar. Para hacerlo, tendrá que hacer manualmente lo que `manage.py` hace:

- Añade tu ruta de proyecto a tu PYTHONPATH
- Establecer el DJANGO_SETTINGS_MODULE

```
export PYTHONPATH="/home/me/path/to/your_project"
export DJANGO_SETTINGS_MODULE="your_project.settings"
```

Esto es especialmente útil en un virtualenv donde puede configurar estas variables de entorno en el script `postactivate` .

`django-admin` comando `django-admin` tiene la ventaja de estar disponible donde sea que esté en su sistema de archivos.

Comandos de gestión incorporados

Django viene con una serie de comandos de administración incorporados, que utilizan `python manage.py [command]` o, cuando manage.py tiene derechos + x (ejecutables) simplemente `./manage.py [command]` . Los siguientes son algunos de los más utilizados:

Obtener una lista de todos los comandos disponibles

```
./manage.py help
```

Ejecute su servidor Django en localhost: 8000; esencial para las pruebas locales

```
./manage.py runserver
```

Ejecute una consola de python (o ipython si está instalada) con la configuración de Django de su proyecto precargada (intentar acceder a partes de su proyecto en un terminal de python sin hacer esto fallará).

```
./manage.py shell
```

Cree un nuevo archivo de migración de base de datos en función de los cambios que haya realizado en sus modelos. Ver migraciones

```
./manage.py makemigrations
```

Aplique las migraciones no aplicadas a la base de datos actual.

```
./manage.py migrate
```

Ejecute la suite de prueba de su proyecto. Ver Unidad de Pruebas

```
./manage.py test
```

Tome todos los archivos estáticos de su proyecto y colóquelos en la carpeta especificada en `STATIC_ROOT` para que puedan ser servidos en producción.

```
./manage.py collectstatic
```

Permitir crear superusuario.

```
./manage.py createsuperuser
```

Cambia la contraseña de un usuario específico.

```
./manage.py changepassword username
```

Lista completa de comandos disponibles

Lea Comandos de gestión en línea: https://riptutorial.com/es/django/topic/1661/comandos-de-gestion

Capítulo 9: Cómo restablecer las migraciones django

Introducción

A medida que desarrolla una aplicación Django, puede haber situaciones en las que puede ahorrar mucho tiempo simplemente limpiando y reiniciando sus migraciones.

Examples

Restablecimiento de la migración de Django: eliminar la base de datos existente y migrar como nueva

Eliminar / Eliminar su base de datos Si está utilizando SQLite para su base de datos, simplemente elimine este archivo. Si está utilizando MySQL / Postgres o cualquier otro sistema de base de datos, deberá abandonar la base de datos y luego volver a crear una base de datos nueva.

Ahora deberá eliminar todos los archivos de migraciones, excepto el archivo "init.py" ubicado dentro de la carpeta de migraciones en la carpeta de su aplicación.

Por lo general, la carpeta de migraciones se encuentra en

```
/your_django_project/your_app/migrations
```

Ahora que ha eliminado la base de datos y el archivo de migraciones, simplemente ejecute los siguientes comandos como si fuera a migrar la primera vez que configure el proyecto django.

```
python manage.py makemigrations
python manage.py migrate
```

Lea Cómo restablecer las migraciones django en línea:
https://riptutorial.com/es/django/topic/9513/como-restablecer-las-migraciones-django

Capítulo 10: Configuración de la base de datos

Examples

MySQL / MariaDB

Django soporta MySQL 5.5 y superior.

Asegúrese de tener algunos paquetes instalados:

```
$ sudo apt-get install mysql-server libmysqlclient-dev
$ sudo apt-get install python-dev python-pip              # for python 2
$ sudo apt-get install python3-dev python3-pip            # for python 3
```

Además de uno de los controladores MySQL de Python (`mysqlclient` la opción recomendada para Django):

```
$ pip install mysqlclient    # python 2 and 3
$ pip install MySQL-python   # python 2
$ pip install pymysql        # python 2 and 3
```

> La codificación de la base de datos no puede ser configurada por Django, pero debe configurarse en el nivel de la base de datos. Busque el `default-character-set` en my.cnf (o `/etc/mysql/mariadb.conf/*.cnf`) y configure la codificación:

```
[mysql]
#default-character-set = latin1    #default on some systems.
#default-character-set = utf8mb4   #default on some systems.
default-character-set = utf8

...
[mysqld]
#character-set-server  = utf8mb4
#collation-server      = utf8mb4_general_ci
character-set-server  = utf8
collation-server      = utf8_general_ci
```

Configuración de la base de datos para MySQL o MariaDB

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'DB_NAME',
        'USER': 'DB_USER',
        'PASSWORD': 'DB_PASSWORD',
        'HOST': 'localhost',   # Or an IP Address that your database is hosted on
```

```
        'PORT': '3306',
        #optional:
        'OPTIONS': {
            'charset' : 'utf8',
            'use_unicode' : True,
             'init_command': 'SET '
                'storage_engine=INNODB,'
                'character_set_connection=utf8,'
                'collation_connection=utf8_bin'
                #'sql_mode=STRICT_TRANS_TABLES,'    # see note below
                #'SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED',
        },
        'TEST_CHARSET': 'utf8',
        'TEST_COLLATION': 'utf8_general_ci',
    }
}
```

Si está utilizando el conector MySQL de Oracle, su línea `ENGINE` debería tener este aspecto:

```
'ENGINE': 'mysql.connector.django',
```

Cuando cree una base de datos, asegúrese de que para especificar la codificación y la intercalación:

```
CREATE DATABASE mydatabase CHARACTER SET utf8 COLLATE utf8_bin
```

> Desde MySQL 5.7 en adelante y en nuevas instalaciones de MySQL 5.6, el valor predeterminado de la opción **sql_mode** contiene **STRICT_TRANS_TABLES** . Esa opción convierte las advertencias en errores cuando los datos se truncan al insertarlos. Django recomienda encarecidamente la activación de un *modo estricto* para MySQL para evitar la pérdida de datos (STRICT_TRANS_TABLES o STRICT_ALL_TABLES). Para habilitar agregar a */etc/my.cnf* `sql-mode = STRICT_TRANS_TABLES`

PostgreSQL

Asegúrese de tener algunos paquetes instalados:

```
sudo apt-get install libpq-dev
pip install psycopg2
```

Configuraciones de la base de datos para PostgreSQL:

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'myprojectDB',
        'USER': 'myprojectuser',
        'PASSWORD': 'password',
        'HOST': '127.0.0.1',
        'PORT': '5432',
```

```
    }
}
```

En versiones anteriores también puede usar el alias `django.db.backends.postgresql_psycopg2` .

Cuando uses Postresql tendrás acceso a algunas características adicionales:

Modelfields:

```
ArrayField         # A field for storing lists of data.
HStoreField        # A field for storing mappings of strings to strings.
JSONField          # A field for storing JSON encoded data.
IntegerRangeField  # Stores a range of integers
BigIntegerRangeField # Stores a big range of integers
FloatRangeField    # Stores a range of floating point values.
DateTimeRangeField # Stores a range of timestamps
```

sqlite

sqlite es el predeterminado para Django. *No debe utilizarse en producción ya que suele ser lento.*

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db/development.sqlite3',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    },
}
```

Accesorios

Los accesorios son datos iniciales para la base de datos. La forma más sencilla cuando ya tiene algunos datos existentes es usar el comando `dumpdata`

```
 ./manage.py dumpdata > databasedump.json                # full database
 ./manage.py dumpdata myapp > databasedump.json          # only 1 app
 ./manage.py dumpdata myapp.mymodel > databasedump.json  # only 1 model (table)
```

Esto creará un archivo json que puede ser importado nuevamente usando

```
./manage.py loaddata databasedump.json
```

Al usar el `loadddata` sin especificar un archivo, Django buscará una carpeta de `fixtures` en su aplicación o la lista de directorios provista en `FIXTURE_DIRS` en la configuración, y usará su contenido en su lugar.

```
/myapp
   /fixtures
        myfixtures.json
        morefixtures.xml
```

Los posibles formatos de archivo son: `JSON, XML or YAML`

Ejemplo de JSON de fixtures:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

Ejemplo de YAML:

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

Ejemplo XML de accesorios:

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="1" model="myapp.person">
        <field type="CharField" name="first_name">John</field>
        <field type="CharField" name="last_name">Lennon</field>
    </object>
    <object pk="2" model="myapp.person">
        <field type="CharField" name="first_name">Paul</field>
        <field type="CharField" name="last_name">McCartney</field>
    </object>
</django-objects>
```

Motor Django Cassandra

- Instala pip: `$ pip install django-cassandra-engine`
- Agregue Getting Started to INSTALLED_APPS en su archivo settings.py: `INSTALLED_APPS = ['django_cassandra_engine']`
- Configuración de BASES DE DATOS Cange Standart:

  Standart

```
DATABASES = {
    'default': {
        'ENGINE': 'django_cassandra_engine',
        'NAME': 'db',
        'TEST_NAME': 'test_db',
        'HOST': 'db1.example.com,db2.example.com',
        'OPTIONS': {
            'replication': {
                'strategy_class': 'SimpleStrategy',
                'replication_factor': 1
            }
        }
    }
}
```

  Cassandra crea nuevo usuario cqlsh:

```
DATABASES = {
'default': {
    'ENGINE': 'django_cassandra_engine',
    'NAME': 'db',
    'TEST_NAME': 'test_db',
    'USER_NAME'='cassandradb',
    'PASSWORD'= '123cassandra',
    'HOST': 'db1.example.com,db2.example.com',
    'OPTIONS': {
        'replication': {
            'strategy_class': 'SimpleStrategy',
            'replication_factor': 1
        }
    }
}
```

}

Lea Configuración de la base de datos en línea:
https://riptutorial.com/es/django/topic/4933/configuracion-de-la-base-de-datos

Capítulo 11: CRUD en Django

Examples

** Ejemplo de CRUD más simple **

Si encuentra estos pasos desconocidos, considere comenzar aquí . Tenga en cuenta que estos pasos provienen de la documentación de desbordamiento de pila.

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

myproject / settings.py Instala la aplicación

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

Cree un archivo llamado `urls.py` dentro del directorio **myapp** y `urls.py` con la siguiente vista.

```
from django.conf.urls import url
from myapp import views

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

Actualice el otro archivo `urls.py` con el siguiente contenido.

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from myapp import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^myapp/', include('myapp.urls')),
    url(r'^admin/', admin.site.urls),
]
```

Crea una carpeta llamada `templates` dentro del directorio **myapp** . Luego cree un archivo llamado `index.html` dentro del directorio de **plantillas** . Rellénalo con el siguiente contenido.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

También necesitamos una vista para mostrar **index.html** que podemos crear editando el archivo **views.py de esta** manera:

```
from django.shortcuts import render, redirect

# Create your views here.
def index(request):
    return render(request, 'index.html', {})
```

Ahora tienes la base con la que vas a trabajar. El siguiente paso es crear un modelo. Este es el ejemplo más simple posible, así que en su carpeta **models.py** agregue el siguiente código.

```
from __future__ import unicode_literals

from django.db import models

# Create your models here.
class Name(models.Model):
    name_value = models.CharField(max_length=100)

    def __str__(self): # if Python 2 use __unicode__
        return self.name_value
```

Esto crea un modelo de un objeto Nombre que agregaremos a la base de datos con los siguientes comandos desde la línea de comandos.

```
python manage.py createsuperuser
python manage.py makemigrations
python manage.py migrate
```

Deberías ver algunas operaciones realizadas por Django. Estos configuran las tablas y crean un superusuario que puede acceder a la base de datos de administración desde una vista de administración potenciada por Django. Hablando de eso, permite registrar nuestro nuevo modelo con la vista de administrador. Vaya a **admin.py** y agregue el siguiente código.

```
from django.contrib import admin
from myapp.models import Name
# Register your models here.

admin.site.register(Name)
```

De vuelta en la línea de comandos, ahora puede activar el servidor con el comando de `python manage.py runserver` . Debería poder visitar http: // localhost: 8000 / y ver su aplicación. Luego, vaya a http: // localhost: 8000 / admin para poder agregar un nombre a su proyecto. Inicie sesión y agregue un Nombre debajo de la tabla MYAPP, lo mantuvimos simple para el ejemplo, así que asegúrese de que tenga menos de 100 caracteres.

Para acceder al nombre necesitas mostrarlo en algún lugar. Edite la función de índice en **views.py** para obtener todos los objetos de Nombre de la base de datos.

```
from django.shortcuts import render, redirect
from myapp.models import Name

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()
    context_dict = {'names_from_context': names_from_db}
    return render(request, 'index.html', context_dict)
```

Ahora edite el archivo **index.html** a lo siguiente.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

Eso demuestra la lectura en CRUD. Dentro del directorio **myapp** crea un archivo forms.py. Agregue el siguiente código:

```
from django import forms
from myapp.models import Name

class NameForm(forms.ModelForm):
    name_value = forms.CharField(max_length=100, help_text = "Enter a name")

    class Meta:
        model = Name
        fields = ('name_value',)
```

Actualice el **index.html** de la siguiente manera:

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <form id="name_form" method="post" action="/">
        {% csrf_token %}
        {% for field in form.visible_fields %}
            {{ field.errors }}
            {{ field.help_text }}
            {{ field }}
        {% endfor %}
        <input type="submit" name="submit" value="Create">
    </form>
</body>
</html>
```

A continuación actualiza el **views.py** de la siguiente manera:

```
from django.shortcuts import render, redirect
from myapp.models import Name
from myapp.forms import NameForm

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()

    form = NameForm()

    context_dict = {'names_from_context': names_from_db, 'form': form}

    if request.method == 'POST':
        form = NameForm(request.POST)

        if form.is_valid():
            form.save(commit=True)
            return render(request, 'index.html', context_dict)
        else:
            print(form.errors)

    return render(request, 'index.html', context_dict)
```

Reinicie su servidor y ahora debería tener una versión de trabajo de la aplicación con la C en la

creación completada.

TODO agregar actualización y eliminar

Lea CRUD en Django en línea: https://riptutorial.com/es/django/topic/7317/crud-en-django

Capítulo 12: Depuración

Observaciones

Pdb

Pdb también puede imprimir todas las variables existentes en el ámbito global o local, escribiendo `globals()` o `locals()` en el indicador (Pdb) respectivamente.

Examples

Usando el depurador de Python (Pdb)

La herramienta de depuración más básica de Django es pdb , una parte de la biblioteca estándar de Python.

Script de vista inicial

Examinemos un script simple de `views.py` :

```
from django.http import HttpResponse


def index(request):
    foo = 1
    bar = 0

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Comando de consola para ejecutar el servidor:

```
python manage.py runserver
```

Es obvio que Django lanzaría un `ZeroDivisionError` cuando intentas cargar una página de índice, pero si pretendemos que el error está muy metido en el código, podría ser realmente desagradable.

Estableciendo un punto de ruptura

Afortunadamente, podemos establecer un *punto* de *interrupción* para rastrear ese error:

```
from django.http import HttpResponse

# Pdb import
import pdb
```

```
def index(request):
    foo = 1
    bar = 0

    # This is our new breakpoint
    pdb.set_trace()

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Comando de consola para ejecutar servidor con pdb:

```
python -m pdb manage.py runserver
```

Ahora, en la página de carga, el punto de interrupción activará (pdb) en el shell, lo que también bloqueará su navegador en estado pendiente.

Depuración con shell pdb

Es hora de depurar esa vista interactuando con el script a través de shell:

```
> ../views.py(12)index()
-> bug = foo/bar
# input 'foo/bar' expression to see division results:
(Pdb) foo/bar
*** ZeroDivisionError: division by zero
# input variables names to check their values:
(Pdb) foo
1
(Pdb) bar
0
# 'bar' is a source of the problem, so if we set it's value > 0...
(Pdb) bar = 1
(Pdb) foo/bar
1.0
# exception gone, ask pdb to continue execution by typing 'c':
(Pdb) c
[03/Aug/2016 10:50:45] "GET / HTTP/1.1" 200 111
```

En la última línea vemos que nuestra vista devolvió una respuesta `OK` y se ejecutó como debería.

Para detener el bucle pdb, simplemente ingrese `q` en un shell.

Usando la barra de herramientas de depuración de Django

Primero, necesitas instalar django-debug-toolbar :

```
pip install django-debug-toolbar
```

settings.py :

A continuación, inclúyalo a las aplicaciones instaladas del proyecto, pero tenga cuidado, siempre

es una buena práctica usar un archivo `settings.py` diferente para aplicaciones de desarrollo y middlewares como la barra de herramientas de depuración:

```
# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]

MIDDLEWARE += ['debug_toolbar.middleware.DebugToolbarMiddleware']
```

La barra de herramientas de depuración también se basa en archivos estáticos, por lo que la aplicación apropiada también debería incluirse:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.staticfiles',
    # ...
]

STATIC_URL = '/static/'

# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]
```

En algunos casos, también se requiere establecer `INTERNAL_IPS` en `settings.py` :

```
INTERNAL_IPS = ('127.0.0.1', )
```

urls.py :

En `urls.py` , como sugiere la documentación oficial, el siguiente fragmento de código debe habilitar el enrutamiento de la barra de herramientas de depuración:

```
if settings.DEBUG and 'debug_toolbar' in settings.INSTALLED_APPS:
    import debug_toolbar
    urlpatterns += [
        url(r'^__debug__/', include(debug_toolbar.urls)),
    ]
```

Recoger la estática de la barra de herramientas después de la instalación:

```
python manage.py collectstatic
```

Eso es todo, la barra de herramientas de depuración aparecerá en las páginas de su proyecto, proporcionando información útil sobre el tiempo de ejecución, SQL, archivos estáticos, señales, etc.

HTML:

Además, `django-debug-toolbar` requiere un *tipo* de *Contenido* de `text/html` , `<html>` y `<body>` etiquetas para representarse correctamente.

En caso de que esté seguro de que ha configurado todo correctamente, pero la barra de herramientas de depuración aún no está renderizada: use esta solución "nuclear" para intentar averiguarlo.

Usando "afirmar falso"

Durante el desarrollo, insertando la siguiente línea en su código:

```
assert False, value
```

hará que django genere un `AssertionError` con el valor proporcionado como mensaje de error cuando se ejecute esta línea.

Si esto ocurre en una vista, o en cualquier código llamado desde una vista, y se establece `DEBUG=True` , se mostrará en el navegador una pila completa y detallada con mucha información de depuración.

No te olvides de eliminar la línea cuando hayas terminado!

Considere escribir más documentación, pruebas, registro y aserciones en lugar de usar un depurador

La depuración lleva tiempo y esfuerzo.

En lugar de perseguir errores con un depurador, considere dedicar más tiempo a mejorar su código al:

- **Escribir y ejecutar pruebas .** Python y Django tienen grandes marcos de prueba integrados, que pueden usarse para probar su código mucho más rápido que manualmente con un depurador.
- **Escribiendo la documentación adecuada** para sus funciones, clases y módulos. PEP 257 y la Guía de estilo Python de Google proporcionan buenas prácticas para escribir buenas cadenas de documentos.
- **Utilice el registro** para producir resultados de su programa, durante el desarrollo y después de la implementación.
- **Agregue `assert`** a su código en lugares importantes: reduzca la ambigüedad, detecte los problemas a medida que se crean.

Bono: ¡Escriba doctests para combinar documentación y pruebas!

Lea Depuración en línea: https://riptutorial.com/es/django/topic/5072/depuracion

Capítulo 13: Despliegue

Examples

Ejecutando la aplicación Django con Gunicorn

1. Instalar gunicorn

   ```
   pip install gunicorn
   ```

2. Desde la carpeta del proyecto django (la misma carpeta donde reside manage.py), ejecute el siguiente comando para ejecutar el proyecto actual de django con gunicorn

   ```
   gunicorn [projectname].wsgi:application -b 127.0.0.1:[port number]
   ```

   Puede usar la opción `--env` para configurar la ruta de acceso para cargar la configuración

   ```
   gunicorn --env DJANGO_SETTINGS_MODULE=[projectname].settings [projectname].wsgi
   ```

   o ejecutar como proceso daemon usando la opción `-D`

3. Al iniciar con éxito gunicorn, las siguientes líneas aparecerán en la consola

   ```
   Starting gunicorn 19.5.0
   ```

   ```
   Listening at: http://127.0.0.1:[port number] ([pid])
   ```

   .... (otra información adicional sobre el servidor gunicorn)

Desplegando con Heroku

1. Descargar Heroku Toolbelt .

2. Navegue a la raíz de las fuentes de su aplicación Django. Necesitarás tk

3. Escribe `heroku create [app_name]` . Si no das un nombre de aplicación, Heroku generará una aleatoriamente para ti. La URL de su aplicación será `http://[app name].herokuapp.com`

4. Haga un archivo de texto con el nombre `Procfile` . No pongas una extensión al final.

   ```
   web: <bash command to start production server>
   ```

   > Si tiene un proceso de trabajo, puede agregarlo también. Agregue otra línea en el formato: `worker-name: <bash command to start worker>`

5. Añadir un requisito.txt.

- Si está utilizando un entorno virtual, ejecute `pip freeze > requirements.txt`
- De lo contrario, *consigue un entorno virtual!* . También puede listar manualmente los paquetes de Python que necesita, pero eso no se tratará en este tutorial.

6. **¡Es tiempo de despliegue!**

    1. `git push heroku master`

       Heroku necesita un repositorio git o una carpeta de Dropbox para realizar implementaciones. Alternativamente, puede configurar la recarga automática desde un repositorio de GitHub en `heroku.com` , pero no lo cubriremos en este tutorial.

    2. `heroku ps:scale web=1`

       Esto escala el número de web "dynos" a uno. Puedes aprender más sobre dynos aquí.

    3. `heroku open` o navega a `http://app-name.herokuapp.com`

       Consejo: `heroku open` abre la URL de su aplicación heroku en el navegador predeterminado.

7. Añadir **complementos** . Tendrá que configurar su aplicación Django para enlazar con las bases de datos proporcionadas en Heroku como "complementos". Este ejemplo no cubre esto, pero otro ejemplo está en la tubería de implementación de bases de datos en Heroku.

Despliegue remoto simple fabfile.py

Fabric es una biblioteca de Python (2.5-2.7) y una herramienta de línea de comandos para agilizar el uso de SSH para la implementación de aplicaciones o tareas de administración de sistemas. Te permite ejecutar funciones de Python arbitrarias a través de la línea de comandos.

Instalar la tela a través de la `pip install fabric`
Crea `fabfile.py` en tu directorio raíz:

```
#myproject/fabfile.py
from fabric.api import *

@task
def dev():
    # details of development server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def release():
    # details of release server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def run():
    with cd('path/to/your_project/'):
```

```
        with prefix('source ../env/bin/activate'):
        # activate venv, suppose it appear in one level higher
            # pass commands one by one
            run('git pull')
            run('pip install -r requirements.txt')
            run('python manage.py migrate --noinput')
            run('python manage.py collectstatic --noinput')
            run('touch reload.txt')
```

Para ejecutar el archivo, simplemente use el comando `fab` :

```
$ fab dev run  # for release server, `fab release run`
```

Nota: no puede configurar las claves ssh para github y simplemente escriba inicio de sesión y contraseña manualmente, mientras se ejecuta fabfile, lo mismo que con las claves.

Usando Heroku Django Starter Template.

Si planea alojar su sitio web de Django en Heroku, puede comenzar su proyecto usando la plantilla de inicio de Heroku Django:

```
django-admin.py startproject --template=https://github.com/heroku/heroku-django-
template/archive/master.zip --name=Procfile YourProjectName
```

Tiene una configuración lista para producción para archivos estáticos, configuraciones de base de datos, Gunicorn, etc. y mejoras en la funcionalidad de servicio de archivos estáticos de Django a través de WhiteNoise. Esto le ahorrará tiempo, está listo para hospedarse en Heroku, simplemente cree su sitio web en la parte superior de esta plantilla

Para desplegar esta plantilla en Heroku:

```
git init
git add -A
git commit -m "Initial commit"

heroku create
git push heroku master

heroku run python manage.py migrate
```

¡Eso es!

Instrucciones de despliegue de Django. Nginx + Gunicorn + Supervisor en Linux (Ubuntu)

Tres herramientas básicas.

1. nginx: servidor HTTP de alto rendimiento, de código abierto y gratuito, y proxy inverso, con alto rendimiento;
2. gunicorn - 'Green Unicorn' es un servidor HTTP WSGI de Python para UNIX (necesario para

administrar su servidor);
3. supervisor: un sistema cliente / servidor que permite a sus usuarios monitorear y controlar una serie de procesos en sistemas operativos similares a UNIX. Se utiliza cuando la aplicación o el sistema se bloquea, reinicia la cámara de django / celery / apio, etc .;

Para simplificarlo, supongamos que su aplicación se encuentra en este directorio: `/home/root/app/src/` y usaremos un usuario `root` (pero debe crear un usuario separado para su aplicación). También nuestro entorno virtual se ubicará en `/home/root/app/env/` path.

NGINX

Vamos a empezar con nginx. Si nginx no está ya en la máquina, instálelo con `sudo apt-get install nginx` . Más adelante, tendrá que crear un nuevo archivo de configuración en su directorio nginx `/etc/nginx/sites-enabled/yourapp.conf` . Si hay un archivo llamado `default.conf` , elimínelo.

Bellow código a un archivo conf nginx, que intentará ejecutar su servicio con el uso de archivo de socket; Más adelante habrá una configuración de gunicorn. El archivo de socket se usa aquí para comunicarse entre nginx y gunicorn. También se puede hacer utilizando puertos.

```
# your application name; can be whatever you want
upstream yourappname {
    server        unix:/home/root/app/src/gunicorn.sock fail_timeout=0;
}

server {
    # root folder of your application
    root        /home/root/app/src/;

    listen        80;
    # server name, your main domain, all subdomains and specific subdomains
    server_name   yourdomain.com *.yourdomain.com somesubdomain.yourdomain.com

    charset       utf-8;

    client_max_body_size                        100m;

    # place where logs will be stored;
    # folder and files have to be already located there, nginx will not create
    access_log        /home/root/app/src/logs/nginx-access.log;
    error_log         /home/root/app/src/logs/nginx-error.log;

    # this is where your app is served (gunicorn upstream above)
    location / {
        uwsgi_pass  yourappname;
        include     uwsgi_params;
    }

    # static files folder, I assume they will be used
    location /static/ {
        alias         /home/root/app/src/static/;
    }

    # media files folder
    location /media/ {
        alias         /home/root/app/src/media/;
```

```
    }

}
```

GUNICORN

Ahora nuestro script GUNICORN, que será responsable de ejecutar la aplicación django en el servidor. Lo primero es instalar gunicorn en un entorno virtual con `pip install gunicorn` .

```
#!/bin/bash

ME="root"
DJANGODIR=/home/root/app/src # django app dir
SOCKFILE=/home/root/app/src/gunicorn.sock # your sock file - do not create it manually
USER=root
GROUP=webapps
NUM_WORKERS=3
DJANGO_SETTINGS_MODULE=yourapp.yoursettings
DJANGO_WSGI_MODULE=yourapp.wsgi
echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR

source /home/root/app/env/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Gunicorn
# Programs meant to be run under supervisor should not daemonize themselves (do not use --
daemon)
exec /home/root/app/env/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name root \
  --workers $NUM_WORKERS \
  --user=$USER --group=$GROUP \
  --bind=unix:$SOCKFILE \
  --log-level=debug \
  --log-file=-
```

para poder ejecutar el script de inicio gunicorn tiene que tener el modo de ejecución habilitado para que

```
sudo chmod u+x /home/root/app/src/gunicorn_start
```

ahora podrá iniciar su servidor gunicorn con solo usar `./gunicorn_start`

SUPERVISOR

Como se dijo al principio, queremos que nuestra aplicación se reinicie cuando un supervisor falla.

Si el supervisor aún no está en el servidor, instale con `sudo apt-get install supervisor` .

Al principio instalar supervisor. Luego cree un archivo `.conf` en su directorio principal `/etc/supervisor/conf.d/your_conf_file.conf`

contenido del archivo de configuración:

```
[program:yourappname]
command = /home/root/app/src/gunicorn_start
user = root
stdout_logfile = /home/root/app/src/logs/gunicorn_supervisor.log
redirect_stderr = true
```

Breve `[program:youappname]` se requiere `[program:youappname]` al principio, será nuestro identificador. también `stdout_logfile` es un archivo donde se almacenarán los registros, tanto de acceso como de errores.

Una vez hecho esto, tenemos que decirle a nuestro supervisor que acabamos de agregar un nuevo archivo de configuración. Para hacerlo, hay diferentes procesos para diferentes versiones de Ubuntu.

Para la `Ubuntu version 14.04 or lesser` , simplemente ejecute esos comandos:

`sudo supervisorctl reread` -> vuelve a leer todos los archivos de configuración dentro del catálogo de supervisor, esto debería imprimirse: nombre de **aplicación: disponible**

`sudo supervisorctl update` -> actualiza supervisor a los archivos de configuración recién agregados; debe imprimir su nombre de **aplicación: grupo de proceso agregado**

Para `Ubuntu 16.04` Ejecutar:

```
sudo service supervisor restart
```

y para comprobar si su aplicación se está ejecutando correctamente, simplemente ejecute

`sudo supervisorctl status yourappname`

Esto debería mostrar:

`yourappname RUNNING pid 18020, uptime 0:00:50`

Para obtener una demostración en vivo de este procedimiento, navegue este **video** .

Despliegue localmente sin configurar apache / nginx

La forma recomendada de implementación de producción requiere el uso de Apache / Nginx para servir el contenido estático. Por lo tanto, cuando `DEBUG` es falsa estática y los contenidos de los medios no se cargan. Sin embargo, podemos cargar el contenido estático en la implementación sin tener que configurar el servidor Apache / Nginx para nuestra aplicación usando:

```
python manage.py runserver --insecure
```

Esto solo está destinado a la implementación local (por ejemplo, LAN) y nunca debe usarse en producción y solo está disponible si la aplicación `staticfiles` está en la configuración `INSTALLED_APPS` su proyecto.

Lea Despliegue en línea: https://riptutorial.com/es/django/topic/2792/despliegue

Capítulo 14: Django desde la línea de comandos.

Observaciones

Si bien Django es principalmente para aplicaciones web, tiene un ORM potente y fácil de usar que también se puede usar para aplicaciones de línea de comandos y scripts. Hay dos enfoques diferentes que pueden ser utilizados. El primero es crear un comando de administración personalizado y el segundo es inicializar el entorno Django al inicio de su script.

Examples

Django desde la línea de comandos.

Suponiendo que ha configurado un proyecto django y que el archivo de configuración se encuentra en una aplicación llamada main, esta es la forma en que inicializa su código

```
import os, sys

# Setup environ
sys.path.append(os.getcwd())
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "main.settings")

# Setup django
import django
django.setup()

# rest of your imports go here

from main.models import MyModel

# normal python code that makes use of Django models go here

for obj in MyModel.objects.all():
    print obj
```

Lo anterior puede ser ejecutado como

```
python main/cli.py
```

Lea Django desde la línea de comandos. en línea:
https://riptutorial.com/es/django/topic/5848/django-desde-la-linea-de-comandos-

Capítulo 15: Django Rest Framework

Examples

Barebones simples API de solo lectura

Suponiendo que tenga un modelo que se parece al siguiente, iniciaremos una ejecución con una simple API de **solo lectura** basada en barebones basada en Django REST Framework ("DRF").

modelos.py

```
class FeedItem(models.Model):
    title = models.CharField(max_length=100, blank=True)
    url = models.URLField(blank=True)
    style = models.CharField(max_length=100, blank=True)
    description = models.TextField(blank=True)
```

El serializador es el componente que tomará toda la información del modelo de Django (en este caso, el artículo de `FeedItem`) y lo convertirá en JSON. Es muy similar a crear clases de formulario en Django. Si tiene alguna experiencia en eso, esto será muy cómodo para usted.

serializers.py

```
from rest_framework import serializers
from . import models

class FeedItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.FeedItem
        fields = ('title', 'url', 'description', 'style')
```

vistas.py

DRF ofrece muchas clases de vistas para manejar una variedad de casos de uso. En este ejemplo, sólo vamos a tener una API **de solo lectura,** por lo que, en lugar de utilizar un enfoque más integral viewset , o un montón de puntos de vista genéricos relacionados, vamos a utilizar una sola subclase de DRF de `ListAPIView` .

El propósito de esta clase es vincular los datos con el serializador y envolverlos todos juntos para un objeto de respuesta.

```
from rest_framework import generics
from . import serializers, models

class FeedItemList(generics.ListAPIView):
    serializer_class = serializers.FeedItemSerializer
    queryset = models.FeedItem.objects.all()
```

urls.py

Asegúrese de apuntar su ruta a su vista DRF.

```
from django.conf.urls import url
from . import views

urlpatterns = [
    ...
    url(r'path/to/api', views.FeedItemList.as_view()),
]
```

Lea Django Rest Framework en línea: https://riptutorial.com/es/django/topic/7341/django-rest-framework

Capítulo 16: Django y redes sociales

Parámetros

Ajuste	Hace
Algunas configuraciones	Prácticas configuraciones básicas que van con Django-Allauth (que uso la mayor parte del tiempo). Para más opciones de configuración, vea Configuraciones.
ACCOUNT_AUTHENTICATION_METHOD (= "username" o "email" o "username_email")	Especifica el método de inicio de sesión que se debe usar, ya sea que el usuario inicie sesión ingresando su nombre de usuario, dirección de correo electrónico o cualquiera de los dos. La configuración de "correo electrónico" requiere ACCOUNT_EMAIL_REQUIRED = Verdadero
ACCOUNT_EMAIL_CONFIRMATION_EXPIRE_DAYS (= 3)	Determina la fecha de vencimiento de los correos de confirmación por correo electrónico (número de días).
ACCOUNT_EMAIL_REQUIRED (= False)	El usuario debe entregar una dirección de correo electrónico al registrarse. Esto va en tándem con la configuración `ACCOUNT_AUTHENTICATION_METHOD`
ACCOUNT_EMAIL_VERIFICATION (= "opcional")	Determina el método de verificación de correo electrónico durante el registro: elija uno de "obligatorio", "opcional" o "ninguno". Cuando se configura en "obligatorio", el usuario no puede iniciar sesión hasta que se verifique la dirección de correo electrónico. Elija "opcional" o "ninguno" para permitir inicios de sesión con una dirección de correo electrónico no verificada. En el caso de "opcional", el correo de verificación de correo electrónico aún se envía, mientras que en el caso de "ninguno" no se envían correos de verificación de correo electrónico.
ACCOUNT_LOGIN_ATTEMPTS_LIMIT (= 5)	Número de intentos de inicio de sesión fallidos. Cuando se excede este número, el

Ajuste	Hace
	usuario tiene prohibido iniciar sesión durante los segundos ACCOUNT_LOGIN_ATTEMPTS_TIMEOUT especificados. Si bien esto protege la vista de inicio de sesión de Allauth, no protege el inicio de sesión del administrador de Django para que no sea forzado.
ACCOUNT_LOGOUT_ON_PASSWORD_CHANGE (= Falso)	Determina si el usuario se desconecta automáticamente o no después de cambiar o configurar su contraseña.
SOCIALACCOUNT_PROVIDERS (= dict)	Diccionario que contiene la configuración específica del proveedor.

Examples

Manera fácil: python-social-auth

python-social-auth es un marco que simplifica el mecanismo de autenticación y autorización social. Contiene muchos backends sociales (Facebook, Twitter, Github, LinkedIn, etc.)

INSTALAR

Primero necesitamos instalar el paquete python-social-auth con

```
pip install python-social-auth
```

o descargue el código desde github. Ahora es un buen momento para agregar esto a su `requirements.txt` archivo.

Configurando ajustes.py

En la configuración.py agregue:

```
INSTALLED_APPS = (
    ...
    'social.apps.django_app.default',
    ...
)
```

CONFIGURACIÓN DE BACKENDS

AUTHENTICATION_BACKENDS contiene los backends que usaremos, y solo tenemos que poner lo que necesitamos.

```
AUTHENTICATION_BACKENDS = (
    'social.backends.open_id.OpenIdAuth',
    'social.backends.google.GoogleOpenId',
    'social.backends.google.GoogleOAuth2',
    'social.backends.google.GoogleOAuth',
    'social.backends.twitter.TwitterOAuth',
    'social.backends.yahoo.YahooOpenId',
    ...
    'django.contrib.auth.backends.ModelBackend',
)
```

Es posible que la `settings.py` proyecto.py todavía no tenga un campo `AUTHENTICATION_BACKENDS` . Si ese es el caso agrega el campo. Asegúrese de no perderse `'django.contrib.auth.backends.ModelBackend',` ya que maneja el inicio de sesión por nombre de usuario / contraseña.

Si usamos por ejemplo Facebook y Linkedin Backends necesitamos agregar las claves API

```
SOCIAL_AUTH_FACEBOOK_KEY = 'YOURFACEBOOKKEY'
SOCIAL_AUTH_FACEBOOK_SECRET = 'YOURFACEBOOKSECRET'
```

y

```
SOCIAL_AUTH_LINKEDIN_KEY = 'YOURLINKEDINKEY'
SOCIAL_AUTH_LINKEDIN_SECRET = 'YOURLINKEDINSECRET'
```

Nota : Puede obtener las claves nedded en los desarrolladores de Facebook y en los desarrolladores de Linkedin y aquí puede ver la lista completa y su forma respectiva de especificar la clave de la API y el secreto de la clave.

Nota sobre las claves secretas: las claves secretas deben mantenerse secretas. Aquí hay una explicación de desbordamiento de pila que es útil. Este tutorial es útil para aprender sobre variables ambientales.

TEMPLATE_CONTEXT_PROCESSORS ayudará a las redirecciones, backends y otras cosas, pero al principio solo necesitamos estos:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'social.apps.django_app.context_processors.backends',
    'social.apps.django_app.context_processors.login_redirect',
    ...
)
```

En Django 1.8, la configuración de `TEMPLATE_CONTEXT_PREPROCESSORS` como se muestra arriba está en desuso. Si este es el caso para ti, lo agregarás dentro del dictado de `TEMPLATES` . El tuyo debe verse algo similar a esto:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, "templates")],
```

```
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'social.apps.django_app.context_processors.backends',
                'social.apps.django_app.context_processors.login_redirect',
            ],
        },
    },
]
```

USANDO UN USUARIO PERSONALIZADO

Si está utilizando un modelo de usuario personalizado y desea asociarse con él, simplemente agregue la siguiente línea (aún en **settings.py**)

```
SOCIAL_AUTH_USER_MODEL = 'somepackage.models.CustomUser'
```

`CustomUser` es un modelo que hereda o `CustomUser` del usuario predeterminado.

Configurando urls.py

```
# if you haven't imported inlcude make sure you do so at the top of your file
from django.conf.urls import url, include

urlpatterns = patterns('',
    ...
    url('', include('social.apps.django_app.urls', namespace='social'))
    ...
)
```

Luego necesitamos sincronizar la base de datos para crear los modelos necesarios:

```
./manage.py migrate
```

¡Finalmente podemos jugar!

en alguna plantilla necesitas agregar algo como esto:

```
    <a href="{% url 'social:begin' 'facebook' %}?next={{ request.path }}">Login with
Facebook</a>
    <a href="{% url 'social:begin' 'linkedin' %}?next={{ request.path }}">Login with
Linkedin</a>
```

Si usa otro backend, simplemente cambie 'facebook' por el nombre de backend.

Desconectando usuarios

Una vez que haya iniciado sesión en los usuarios, es probable que desee crear la funcionalidad para volver a cerrarlos. En alguna plantilla, probablemente cerca de donde se mostró la plantilla

de inicio de sesión, agregue la siguiente etiqueta:

```
<a href="{% url 'logout' %}">Logout</a>
```

o

```
<a href="/logout">Logout</a>
```

`urls.py` editar su archivo `urls.py` con un código similar a:

```
url(r'^logout/$', views.logout, name='logout'),
```

Por último, edita tu archivo views.py con un código similar a:

```
def logout(request):
    auth_logout(request)
    return redirect('/')
```

Usando Django Allauth

Para todos mis proyectos, Django-Allauth sigue siendo uno que es fácil de configurar y viene de la caja con muchas características que incluyen pero no se limitan a:

- Unas 50+ autenticaciones de redes sociales.
- Mezclar registro de cuentas locales y sociales.
- Múltiples cuentas sociales
- Registro instantáneo opcional para cuentas sociales, sin preguntas
- Gestión de la dirección de correo electrónico (varias direcciones de correo electrónico, configuración de un primario)
- Contraseña olvidada flujo Correo electrónico flujo de verificación de dirección

Si está interesado en ensuciarse las manos, Django-Allauth se sale del camino, con configuraciones adicionales para ajustar el proceso y el uso de su sistema de autenticación.

Los pasos a continuación asumen que estás usando Django 1.10+

Pasos de configuración:

```
pip install django-allauth
```

En su archivo `settings.py` , realice los siguientes cambios:

```
# Specify the context processors as follows:
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
```

```
                # Already defined Django-related contexts here

                # `allauth` needs this from django. It is there by default,
                # unless you've devilishly taken it away.
                'django.template.context_processors.request',
            ],
        },
    },
]

AUTHENTICATION_BACKENDS = (
    # Needed to login by username in Django admin, regardless of `allauth`
    'django.contrib.auth.backends.ModelBackend',

    # `allauth` specific authentication methods, such as login by e-mail
    'allauth.account.auth_backends.AuthenticationBackend',
)

INSTALLED_APPS = (
# Up here is all your default installed apps from Django

# The following apps are required:
'django.contrib.auth',
'django.contrib.sites',

'allauth',
'allauth.account',
'allauth.socialaccount',

# include the providers you want to enable:
'allauth.socialaccount.providers.google',
'allauth.socialaccount.providers.facebook',
)

# Don't forget this little dude.
SITE_ID = 1
```

Hecho con los cambios en el archivo `settings.py` arriba, muévase al archivo `urls.py` Puede ser su `yourapp/urls.py` o su `ProjectName/urls.py` Normalmente, prefiero el `ProjectName/urls.py`

```
urlpatterns = [
    # other urls here
    url(r'^accounts/', include('allauth.urls')),
    # other urls here
]
```

Simplemente agregando el `include('allauth.urls')` , te da estas urls gratis:

```
^accounts/ ^ ^signup/$ [name='account_signup']
^accounts/ ^ ^login/$ [name='account_login']
^accounts/ ^ ^logout/$ [name='account_logout']
^accounts/ ^ ^password/change/$ [name='account_change_password']
^accounts/ ^ ^password/set/$ [name='account_set_password']
^accounts/ ^ ^inactive/$ [name='account_inactive']
^accounts/ ^ ^email/$ [name='account_email']
^accounts/ ^ ^confirm-email/$ [name='account_email_verification_sent']
^accounts/ ^ ^confirm-email/(?P<key>[-:\w]+)/$ [name='account_confirm_email']
^accounts/ ^ ^password/reset/$ [name='account_reset_password']
```

```
^accounts/ ^ ^password/reset/done/$ [name='account_reset_password_done']
^accounts/ ^ ^password/reset/key/(?P<uidb36>[0-9A-Za-z]+)-(?P<key>.+)/$
[name='account_reset_password_from_key']
^accounts/ ^ ^password/reset/key/done/$ [name='account_reset_password_from_key_done']
^accounts/ ^social/
^accounts/ ^google/
^accounts/ ^twitter/
^accounts/ ^facebook/
^accounts/ ^facebook/login/token/$ [name='facebook_login_by_token']
```

Finalmente, `python ./manage.py migrate` para cometer las migraciones de Django-allauth a la base de datos.

Como de costumbre, para poder iniciar sesión en su aplicación utilizando cualquier red social que haya agregado, deberá agregar los detalles de la cuenta social de la red.

Inicie sesión en el administrador de Django (`localhost:8000/admin`) y en `Social Applications` en el enlace añada su cuenta social.

Es posible que necesite cuentas en cada proveedor de autenticación para obtener detalles para completar en las secciones de Aplicaciones sociales.

Para configuraciones detalladas de lo que puede tener y modificar, vea la página de Configuraciones .

Lea Django y redes sociales en línea: https://riptutorial.com/es/django/topic/4743/django-y-redes-sociales

Capítulo 17: Ejecución de apio con supervisor

Examples

Configuración de apio

APIO

1. Instalación - `pip install django-celery`
2. Añadir
3. Estructura básica del proyecto.

```
- src/
  - bin/celery_worker_start # will be explained later on
  - logs/celery_worker.log
  - stack/__init __.py
  - stack/celery.py
  - stack/settings.py
  - stack/urls.py
  - manage.py
```

4. Agregue el archivo `celery.py` a su `stack/stack/` carpeta.

```
from __future__ import absolute_import
import os
from celery import Celery
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'stack.settings')
from django.conf import settings  # noqa
app = Celery('stack')
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

5. a su `stack/stack/__init__.py` agregue el siguiente código:

```
from __future__ import absolute_import
from .celery import app as celery_app  # noqa
```

6. Cree una tarea y márquela, por ejemplo, como `@shared_task()`

```
@shared_task()
def add(x, y):
    print("x*y={}".format(x*y))
```

7. Trabajador de apio corriendo "a mano":

`celery -A stack worker -l info` si también desea agregar

Supervisor de carrera

1. Crear una secuencia de comandos para iniciar el trabajador de apio. Inserte su script dentro de su aplicación. Por ejemplo: `stack/bin/celery_worker_start`

```
#!/bin/bash

NAME="StackOverflow Project - celery_worker_start"

PROJECT_DIR=/home/stackoverflow/apps/proj/proj/
ENV_DIR=/home/stackoverflow/apps/proj/env/

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd "${PROJECT_DIR}"

if [ -d "${ENV_DIR}" ]
then
    . "${ENV_DIR}bin/activate"
fi

celery -A stack --loglevel='INFO'
```

2. Agregue derechos de ejecución a su script recién creado:

```
chmod u+x bin/celery_worker_start
```

3. Instalar supervisor (omita esta prueba si supervisor ya está instalado)

```
apt-get install supervisor
```

4. Agregue un archivo de configuración para su supervisor con el fin de comenzar su apio. Colóquelo en `/etc/supervisor/conf.d/stack_supervisor.conf`

```
[program:stack-celery-worker]
command = /home/stackoverflow/apps/stack/src/bin/celery_worker_start
user = polsha
stdout_logfile = /home/stackoverflow/apps/stack/src/logs/celery_worker.log
redirect_stderr = true
environment = LANG = en_US.UTF-8,LC_ALL = en_US.UTF-8
numprocs = 1
autostart = true
autorestart = true
startsecs = 10
stopwaitsecs = 600
priority = 998
```

5. Vuelva a leer y actualizar supervisor

```
sudo supervisorctl reread
    stack-celery-worker: available
sudo supervisorctl update
```

```
    stack-celery-worker: added process group
```

6. Comandos básicos

```
sudo supervisorctl status stack-celery-worker
    stack-celery-worker      RUNNING    pid 18020, uptime 0:00:50
sudo supervisorctl stop stack-celery-worker
    stack-celery-worker: stopped
sudo supervisorctl start stack-celery-worker
    stack-celery-worker: started
sudo supervisorctl restart stack-celery-worker
    stack-celery-worker: stopped
    stack-celery-worker: started
```

Apio + RabbitMQ con Supervisor

El apio requiere un corredor para manejar el paso de mensajes. Utilizamos RabbitMQ porque es fácil de configurar y está bien soportado.

Instale rabbitmq usando el siguiente comando

```
sudo apt-get install rabbitmq-server
```

Una vez que se complete la instalación, cree un usuario, agregue un host virtual y configure los permisos.

```
sudo rabbitmqctl add_user myuser mypassword
sudo rabbitmqctl add_vhost myvhost
sudo rabbitmqctl set_user_tags myuser mytag
sudo rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

Para iniciar el servidor:

```
sudo rabbitmq-server
```

Podemos instalar el apio con pip:

```
pip install celery
```

En su archivo settings.py de Django, la URL de su agente se vería como

```
BROKER_URL = 'amqp://myuser:mypassword@localhost:5672/myvhost'
```

Ahora empieza el trabajador del apio.

```
celery -A your_app worker -l info
```

Este comando inicia a un trabajador de apio para ejecutar cualquier tarea definida en su aplicación django.

Supervisor es un programa de Python que le permite controlar y seguir ejecutando cualquier proceso de Unix. También puede reiniciar procesos estrellados. Lo usamos para asegurarnos de que los trabajadores de Apio siempre estén corriendo.

Primero, instala supervisor

```
sudo apt-get install supervisor
```

Cree el archivo your_proj.conf en su supervisor conf.d (/etc/supervisor/conf.d/your_proj.conf):

```
[program:your_proj_celery]
command=/home/your_user/your_proj/.venv/bin/celery --app=your_proj.celery:app worker -l info
directory=/home/your_user/your_proj
numprocs=1
stdout_logfile=/home/your_user/your_proj/logs/celery-worker.log
stderr_logfile=/home/your_user/your_proj/logs/low-worker.log
autostart=true
autorestart=true
startsecs=10
```

Una vez creado y guardado nuestro archivo de configuración, podemos informar al Supervisor de nuestro nuevo programa a través del comando supervisorctl. Primero le pedimos a Supervisor que busque cualquier configuración de programa nueva o modificada en el directorio /etc/supervisor/conf.d con:

```
sudo supervisorctl reread
```

Luego se le indica que promulgue cualquier cambio con:

```
sudo supervisorctl update
```

Una vez que nuestros programas se estén ejecutando, indudablemente habrá un momento en el que deseamos detenernos, reiniciar o ver su estado.

```
sudo supervisorctl status
```

Para reiniciar su instancia de apio:

```
sudo supervisorctl restart your_proj_celery
```

Lea Ejecución de apio con supervisor en línea:
https://riptutorial.com/es/django/topic/7091/ejecucion-de-apio-con-supervisor

Capítulo 18: Enrutadores de base de datos

Examples

Agregar un archivo de enrutamiento de base de datos

Para usar múltiples bases de datos en Django, solo especifique cada una en `settings.py` :

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'django_db_user',
        'PASSWORD': os.environ['LOCAL_DB_PASSWORD']
    },
    'users': {
        'NAME': 'remote_data',
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'remote.host.db',
        'USER': 'remote_user',
        'PASSWORD': os.environ['REMOTE_DB_PASSWORD']
    }
}
```

Utilice un archivo `dbrouters.py` para especificar qué modelos deben operar en qué bases de datos para cada clase de operación de base de datos, por ejemplo, para datos remotos almacenados en `remote_data` , es posible que desee lo siguiente:

```
class DbRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read remote models go to remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def db_for_write(self, model, **hints):
        """
        Attempts to write remote models go to the remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Do not allow relations involving the remote database
        """
        if obj1._meta.app_label == 'remote' or \
```

```
            obj2._meta.app_label == 'remote':
            return False
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Do not allow migrations on the remote database
        """
        if model._meta.app_label == 'remote':
            return False
        return True
```

Finalmente, agregue su `dbrouter.py` a `settings.py` :

```
DATABASE_ROUTERS = ['path.to.DbRouter', ]
```

Especificando diferentes bases de datos en el código.

El `obj.save()` normal `obj.save()` usará la base de datos predeterminada, o si se usa un enrutador de base de datos, usará la base de datos como se especifica en `db_for_write` . Puedes anularlo usando:

```
obj.save(using='other_db')
obj.delete(using='other_db')
```

Del mismo modo, para la lectura:

```
MyModel.objects.using('other_db').all()
```

Lea Enrutadores de base de datos en línea:
https://riptutorial.com/es/django/topic/3395/enrutadores-de-base-de-datos

Capítulo 19: Enrutamiento de URL

Examples

Cómo Django maneja una solicitud

Django maneja una solicitud enrutando la ruta URL entrante a una función de vista. La función de visualización es responsable de devolver una respuesta al cliente que realiza la solicitud. Las diferentes URL son manejadas generalmente por diferentes funciones de vista. Para dirigir la solicitud a una función de vista específica, Django analiza la configuración de su URL (o URLconf para abreviar). La plantilla de proyecto predeterminada define el URLconf en `<myproject>/urls.py` .

Su URLconf debe ser un módulo de Python que define un atributo denominado `urlpatterns` , que es una lista de `django.conf.urls.url()` de `django.conf.urls.url()` . Cada instancia de `url()` debe, como mínimo, definir una expresión regular (una expresión regular) para que coincida con la URL y un objetivo, que es una función de vista o un URLconf diferente. Si un patrón de URL apunta a una función de vista, es una buena idea darle un nombre para hacer referencia fácilmente al patrón más adelante.

Echemos un vistazo a un ejemplo básico:

```
# In <myproject>/urls.py

from django.conf.urls import url

from myapp.views import home, about, blog_detail

urlpatterns = [
    url(r'^$', home, name='home'),
    url(r'^about/$', about, name='about'),
    url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),
]
```

Este URLconf define tres patrones de URL, todos dirigidos a una vista: `home` , `about` y `blog-detail` .

- `url(r'^$', home, name='home'),`

La expresión regular contiene un anclaje de inicio '^', seguido inmediatamente por un anclaje de extremo '$'. Este patrón coincidirá con las solicitudes en las que la ruta de la URL sea una cadena vacía y las `myapp.views` a la vista de `home` definida en `myapp.views` .

- `url(r'^about/$', about, name='about'),`

Esta expresión regular contiene un anclaje de inicio, seguido por la cadena literal `about/` , y el anclaje final. Esto coincidirá con la URL `/about/` y lo dirigirá a la vista `about` . Dado que cada URL no vacía comienza con un `/` , Django corta convenientemente la primera barra para usted.

- `url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),`

Este regex es un poco más complejo. Define el ancla de inicio y la cadena literal `blog/` , como el patrón anterior. La siguiente parte, `(?P<id>\d+)` , se llama un grupo de captura. Un grupo de captura, como sugiere su nombre, captura una parte de la cadena, y Django pasa la cadena capturada como un argumento a la función de vista.

La sintaxis de un grupo de captura es `(?P<name>pattern)` . `name` define el nombre del grupo, que también es el nombre que Django usa para pasar el argumento a la vista. El patrón define qué caracteres coinciden con el grupo.

En este caso, el nombre es `id` , por lo que la función `blog_detail` debe aceptar un parámetro llamado `id` . El patrón es `\d+` . `\d` significa que el patrón solo coincide con caracteres numéricos. `+` significa que el patrón debe coincidir con uno o más caracteres.

Algunos patrones comunes:

Modelo	Usado para	Partidos
`\d+`	carné de identidad	Uno o más caracteres numéricos
`[\w-]+`	babosa	Uno o más caracteres alfanuméricos, guiones bajos o guiones
`[0-9]{4}`	año (largo)	Cuatro números, de cero a nueve.
`[0-9]{2}`	año (corto) mes dia del mes	Dos números, de cero a nueve.
`[^/]+`	segmento de trayectoria	Cualquier cosa excepto una barra

El grupo de captura en el patrón de `blog-detail` está seguido por un literal `/` y el ancla final.

Las URL válidas incluyen:

- `/blog/1/ # passes id='1'`
- `/blog/42/ # passes id='42'`

Las URL inválidas son por ejemplo:

- `/blog/a/ # 'a' does not match '\d'`
- `/blog// # no characters in the capturing group does not match '+'`

Django procesa cada patrón de URL en el mismo orden en que se definen en `urlpatterns` . Esto es importante si varios patrones pueden coincidir con la misma URL. Por ejemplo:

```
urlpatterns = [
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
```

```
]
```

En el URLconf anterior, el segundo patrón no es accesible. El patrón coincidiría con la URL `/blog/overview/` , pero en lugar de llamar a la vista `blog_overview` , la URL primero coincidirá con el patrón de `blog-detail` del `blog-detail` y llamará a la vista `blog_detail` con un argumento `slug='overview'` .

Para asegurarse de que la URL `/blog/overview/` se enrute a la vista `blog_overview` , el patrón se debe colocar sobre el patrón de `blog-detail` del `blog-detail` :

```
urlpatterns = [
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
]
```

Establecer el espacio de nombres de la URL para una aplicación reutilizable (Django 1.9+)

Configure el `app_name` su aplicación para usar automáticamente un espacio de nombres de URL configurando el atributo `app_name` :

```
# In <myapp>/urls.py
from django.conf.urls import url

from .views import overview

app_name = 'myapp'
urlpatterns = [
    url(r'^$', overview, name='overview'),
]
```

Esto establecerá el espacio de nombres de la aplicación en `'myapp'` cuando se incluya en la raíz URLconf>. El usuario de su aplicación reutilizable no necesita hacer ninguna otra configuración que no sea incluir sus URL:

```
# In <myproject>/urls.py
from django.conf.urls import include, url

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

Su aplicación reutilizable ahora puede revertir las URL usando el espacio de nombres de la aplicación:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
```

El URLconf raíz todavía puede establecer un espacio de nombres de instancia con el parámetro de `namespace` :

```
# In <myproject>/urls.py
urlpatterns = [
    url(r'^myapp/', include('myapp.urls', namespace='mynamespace')),
]
```

Tanto el espacio de nombres de la aplicación como el espacio de nombres de la instancia se pueden usar para revertir las URL:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
>>> reverse('mynamespace:overview')
'/myapp/overview/'
```

El espacio de nombres de la instancia se establece de forma predeterminada en el espacio de nombres de la aplicación si no se establece explícitamente.

Lea Enrutamiento de URL en línea: https://riptutorial.com/es/django/topic/3299/enrutamiento-de-url

Capítulo 20: Estructura del proyecto

Examples

Repositorio> Proyecto> Sitio / Conf.

Para un proyecto Django con `requirements` y `deployment tools` bajo control de código fuente. Este ejemplo se basa en los conceptos de las Dos cucharadas de Django . Han publicado una plantilla :

```
repository/
    docs/
    .gitignore
    project/
        apps/
            blog/
                migrations/
                static/ #( optional )
                    blog/
                        some.css
                templates/ #( optional )
                    blog/
                        some.html
                models.py
                tests.py
                admin.py
                apps.py #( django 1.9 and later )
                views.py
            accounts/
                #... ( same as blog )
            search/
                #... ( same as blog )
        conf/
            settings/
                local.py
                development.py
                production.py
            wsgi
            urls.py
        static/
        templates/
    deploy/
        fabfile.py
    requirements/
        base.txt
        local.txt
    README
    AUTHORS
    LICENSE
```

Aquí, las `apps` y las carpetas `conf` contienen `user created applications` y `core configuration folder` para el proyecto respectivamente.

`static` carpetas `static` y de `templates` en el directorio del `project` contienen archivos estáticos y

archivos de `html markup` , respectivamente, que se usan globalmente a lo largo del proyecto.

Y todas las carpetas de aplicaciones de `blog` , `accounts` y `search` también pueden (en su mayoría) contener carpetas `static` y de `templates` .

Espacios de nombres, archivos estáticos y plantillas en aplicaciones django.

`static` carpeta `static` y de `templates` en las aplicaciones también debe contener una carpeta con el nombre de la aplicación `ex. blog` es una convención que se utiliza para evitar la contaminación del espacio de nombres, por lo que hacemos referencia a los archivos como `/blog/base.html` lugar de `/base.html` que proporciona más claridad sobre el archivo al que hacemos referencia y conserva el espacio de nombres.

Ejemplo: `templates` carpeta de `templates` dentro del `blog` y las aplicaciones de `search` contiene un archivo con el nombre `base.html` , y al hacer referencia al archivo en `views` su aplicación se confunde en qué archivo se procesa.

```
(Project Structure)
.../project/
    apps/
        blog/
            templates/
                base.html
        search/
            templates/
                base.html

(blog/views.py)
def some_func(request):
    return render(request, "/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/base.html")

## After creating a folder inside /blog/templates/(blog) ##

(Project Structure)
.../project/
    apps/
        blog/
            templates/
                blog/
                    base.html
        search/
            templates/
                search/
                    base.html

(blog/views.py)
def some_func(request):
    return render(request, "/blog/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/search/base.html")
```

Lea Estructura del proyecto en línea: https://riptutorial.com/es/django/topic/4299/estructura-del-proyecto

Capítulo 21: Etiquetas de plantillas y filtros

Examples

Filtros personalizados

Filtros le permite aplicar una función a una variable. Esta función puede tomar **0** o **1** argumento. Aquí está la sintaxis:

```
{{ variable|filter_name }}
{{ variable|filter_name:argument }}
```

Los filtros se pueden encadenar por lo que esto es perfectamente válido:

```
{{ variable|filter_name:argument|another_filter }}
```

Si se traduce a python, la línea anterior daría algo como esto:

```
print(another_filter(filter_name(variable, argument)))
```

En este ejemplo, escribiremos un filtro personalizado `verbose_name` que se aplique a un Modelo (instancia o clase) o un QuerySet. Devolverá el nombre detallado de un modelo o su nombre completo en plural si el argumento se establece en `True` .

```
@register.filter
def verbose_name(model, plural=False):
    """Return the verbose name of a model.
    `model` can be either:
      - a Model class
      - a Model instance
      - a QuerySet
      - any object refering to a model through a `model` attribute.

    Usage:
      - Get the verbose name of an object
          {{ object|verbose_name }}
      - Get the plural verbose name of an object from a QuerySet
          {{ objects_list|verbose_name:True }}
    """
    if not hasattr(model, '_meta'):
        # handle the case of a QuerySet (among others)
        model = model.model
    opts = model._meta
    if plural:
        return opts.verbose_name_plural
    else:
        return opts.verbose_name
```

Etiquetas simples

La forma más sencilla de definir una etiqueta de plantilla personalizada es usar un `simple_tag` . Estos son muy fáciles de configurar. El nombre de la función será el nombre de la etiqueta (aunque puede anularlo), y los argumentos serán tokens ("palabras" separadas por espacios, excepto los espacios entre comillas). Incluso soporta argumentos de palabras clave.

Aquí hay una etiqueta inútil que ilustrará nuestro ejemplo:

```
{% useless 3 foo 'hello world' foo=True bar=baz.hello|capfirst %}
```

Sean `foo` y `baz` variables de contexto como las siguientes:

```
{'foo': "HELLO", 'baz': {'hello': "world"}}
```

Digamos que queremos que esta etiqueta muy inútil se muestre así:

```
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
```

Tipo de concatenación de argumentos repetida 3 veces (3 es el primer argumento).

Aquí está el aspecto de la implementación de la etiqueta:

```
from django.utils.html import format_html_join

@register.simple_tag
def useless(repeat, *args, **kwargs):
    output = ';'.join(args + ['{}:{}'.format(*item) for item in kwargs.items()])
    outputs = [output] * repeat
    return format_html_join('\n', '{}<br/>', ((e,) for e in outputs))
```

`format_html_join` permite marcar `<br/>` como HTML seguro, pero no el contenido de los `outputs` .

Etiquetas personalizadas avanzadas usando Nodo

A veces, lo que quieres hacer es demasiado complejo para un `filter` o un `simple_tag` . De este modo, deberá crear una función de compilación y un renderizador.

En este ejemplo, crearemos una etiqueta de plantilla `verbose_name` con la siguiente sintaxis:

Ejemplo	Descripción
`{% verbose_name obj %}`	Nombre detallado de un modelo
`{% verbose_name obj 'status' %}`	Nombre detallado del campo "estado"
`{% verbose_name obj plural %}`	Verbose nombre plural de un modelo
`{% verbose_name obj plural capfirst %}`	Nombre verbal en mayúscula plural de un modelo

Ejemplo	Descripción
`{% verbose_name obj 'foo' capfirst %}`	Nombre detallado en mayúscula de un campo
`{% verbose_name obj field_name %}`	Nombre detallado de un campo de una variable
`{% verbose_name obj 'foo'|add:'_bar' %}`	Nombre detallado de un campo "foo_bar"

La razón por la que no podemos hacer esto con una etiqueta simple es que `plural` y `capfirst` no son variables ni cadenas, son "palabras clave". Obviamente, podríamos decidir pasarlos como cadenas `'plural'` y `'capfirst'` , pero puede entrar en conflicto con los campos con estos nombres. ¿ `{% verbose_name obj 'plural' %}` significaría "nombre detallado plural de `obj` " o "nombre detallado de `obj.plural` "?

Primero vamos a crear la función de compilación:

```
@register.tag(name='verbose_name')
def do_verbose_name(parser, token):
    """
    - parser: the Parser object. We will use it to parse tokens into
              nodes such as variables, strings, ...
    - token: the Token object. We will use it to iterate each token
             of the template tag.
    """
    # Split tokens within spaces (except spaces inside quotes)
    tokens = token.split_contents()
    tag_name = tokens[0]
    try:
        # each token is a string so we need to parse it to get the actual
        # variable instead of the variable name as a string.
        model = parser.compile_filter(tokens[1])
    except IndexError:
        raise TemplateSyntaxError(
            "'{}' tag requires at least 1 argument.".format(tag_name))

    field_name = None
    flags = {
        'plural': False,
        'capfirst': False,
    }

    bits = tokens[2:]
    for bit in bits:
        if bit in flags.keys():
            # here we don't need `parser.compile_filter` because we expect
            # 'plural' and 'capfirst' flags to be actual strings.
            if flags[bit]:
                raise TemplateSyntaxError(
                    "'{}' tag only accept one occurrence of '{}' flag".format(
                        tag_name, bit)
                )
            flags[bit] = True
            continue
        if field_name:
            raise TemplateSyntaxError((
                "'{}' tag only accept one field name at most. {} is the second "
                "field name encountered."
```

```
            ).format(tag_name, bit)
        field_name = parser.compile_filter(bit)

    # VerboseNameNode is our renderer which code is given right below
    return VerboseNameNode(model, field_name, **flags)
```

Y ahora el renderizador:

```
class VerboseNameNode(Node):

    def __init__(self, model, field_name=None, **flags):
        self.model = model
        self.field_name = field_name
        self.plural = flags.get('plural', False)
        self.capfirst = flags.get('capfirst', False)

    def get_field_verbose_name(self):
        if self.plural:
            raise ValueError("Plural is not supported for fields verbose name.")
        return self.model._meta.get_field(self.field_name).verbose_name

    def get_model_verbose_name(self):
       if self.plural:
           return self.model._meta.verbose_name_plural
       else:
           return self.model._meta.verbose_name

    def render(self, context):
        """This is the main function, it will be called to render the tag.
        As you can see it takes context, but we don't need it here.
        For instance, an advanced version of this template tag could look for an
        `object` or `object_list` in the context if `self.model` is not provided.
        """
        if self.field_name:
            verbose_name = self.get_field_verbose_name()
        else:
            verbose_name = self.get_model_verbose_name()
        if self.capfirst:
            verbose_name = verbose_name.capitalize()
        return verbose_name
```

Lea Etiquetas de plantillas y filtros en línea: https://riptutorial.com/es/django/topic/1305/etiquetas-de-plantillas-y-filtros

Capítulo 22: Examen de la unidad

Examples

Pruebas - un ejemplo completo

Esto supone que ha leído la documentación sobre cómo iniciar un nuevo proyecto de Django. Supongamos que la aplicación principal de su proyecto se llama td (abreviatura de prueba). Para crear su primera prueba, cree un archivo llamado test_view.py y copie y pegue el siguiente contenido en él.

```
from django.test import Client, TestCase

class ViewTest(TestCase):

    def test_hello(self):
        c = Client()
        resp = c.get('/hello/')
        self.assertEqual(resp.status_code, 200)
```

Puede ejecutar esta prueba por

```
 ./manage.py test
```

¡Y fallará naturalmente! Verás un error similar al siguiente.

```
Traceback (most recent call last):
  File "/home/me/workspace/td/tests_view.py", line 9, in test_hello
    self.assertEqual(resp.status_code, 200)
AssertionError: 200 != 404
```

¿Por qué sucede eso? ¡Porque no hemos definido una vista para eso! Hagamoslo. Cree un archivo llamado views.py y coloque en él el siguiente código

```
from django.http import HttpResponse
def hello(request):
    return HttpResponse('hello')
```

Luego mapéelo a / hello / editando las direcciones URL como sigue:

```
from td import views

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

Ahora ejecuta la prueba otra vez `./manage.py test` otra vez y viola !!

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

Probando Modelos Django Efectivamente

Asumiendo una clase

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('view_author', args=[str(self.id)])


class Book(models.Model):
    author = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    private = models.BooleanField(default=false)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('view_book', args=[str(self.id)])

    def __str__(self):
        return self.name
```

Ejemplos de prueba

```
from django.test import TestCase
from .models import Book, Author

class BaseModelTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseModelTestCase, cls).setUpClass()
        cls.author = Author(name='hawking')
        cls.author.save()
        cls.first_book = Book(author=cls.author, name="short_history_of_time")
        cls.first_book.save()
        cls.second_book = Book(author=cls.author, name="long_history_of_time")
        cls.second_book.save()


class AuthorModelTestCase(BaseModelTestCase):
    def test_created_properly(self):
         self.assertEqual(self.author.name, 'hawking')
         self.assertEqual(True, self.first_book in self.author.book_set.all())

    def test_absolute_url(self):
```

```
        self.assertEqual(self.author.get_absolute_url(), reverse('view_author',
args=[str(self.author.id)]))

class BookModelTestCase(BaseModelTestCase):

    def test_created_properly(self:
        ...
        self.assertEqual(1, len(Book.objects.filter(name__startswith='long'))

    def test_absolute_url(self):
        ...
```

Algunos puntos

- `created_properly` pruebas `created_properly` se utilizan para verificar las propiedades de estado de los modelos de django. Ayudan a detectar situaciones en las que hemos cambiado los valores predeterminados, file_upload_paths, etc.
- `absolute_url` puede parecer trivial, pero he descubierto que me ayudó a evitar algunos errores al cambiar las rutas de URL
- De manera similar, escribo casos de prueba para todos los métodos implementados dentro de un modelo (utilizando objetos `mock` , etc.)
- Al definir una `BaseModelTestCase` común, podemos configurar las relaciones necesarias entre los modelos para garantizar una prueba adecuada.

Finalmente, ante la duda, escribe una prueba. Los cambios de comportamiento triviales se captan prestando atención a los detalles y los fragmentos de código olvidados no terminan causando problemas innecesarios.

Pruebas de control de acceso en Django Views

tl; dr : crea una clase base que define dos objetos de usuario (por ejemplo, `user` y `another_user`). Crea tus otros modelos y define tres instancias de `Client` .

- `self.client` : Representando `user` registrado en el navegador
- `self.another_client` : Representando `another_user` cliente de `another_user`
- `self.unlogged_client` : Representa a una persona no registrada

Ahora acceda a todas sus direcciones URL públicas y privadas desde estos tres objetos de cliente y dicte la respuesta que espera. A continuación, muestro la estrategia para un objeto `Book` que puede ser `private` (propiedad de unos pocos usuarios privilegiados) o `public` (visible para todos).

```
from django.test import TestCase, RequestFactory, Client
from django.core.urlresolvers import reverse

class BaseViewTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseViewTestCase, cls).setUpClass()
        cls.client = Client()
        cls.another_client = Client()
```

```
        cls.unlogged_client = Client()
        cls.user = User.objects.create_user(
                'dummy',password='dummy'
                )
        cls.user.save()
        cls.another_user = User.objects.create_user(
                'dummy2', password='dummy2'
                )
        cls.another_user.save()
        cls.first_book = Book.objects.create(
                name='first',
                private = true
        )
        cls.first_book.readers.add(cls.user)
        cls.first_book.save()
        cls.public_book = Template.objects.create(
                name='public',
                private=False
        )
        cls.public_book.save()


    def setUp(self):
        self.client.login(username=self.user.username, password=self.user.username)
        self.another_client.login(username=self.another_user.username,
password=self.another_user.username)


"""
   Only cls.user owns the first_book and thus only he should be able to see it.
   Others get 403(Forbidden) error
"""
class PrivateBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PrivateBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.first_book.id)})

    def test_user_sees_own_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.first_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_user_cant_see_others_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(403, response.status_code)

    def test_unlogged_user_cant_see_private_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(403, response.status_code)

"""
    Since book is public all three clients should be able to see the book
"""
 class PublicBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PublicBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.public_book.id)})
```

```
    def test_user_sees_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.public_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_another_user_sees_public_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(200, response.status_code)

    def test_unlogged_user_sees_public_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(200, response.status_code)
```

La base de datos y las pruebas

Django usa una configuración especial de la base de datos cuando realiza pruebas, de modo que las pruebas pueden usar la base de datos normalmente, pero se ejecutan por defecto en una base de datos vacía. Los cambios en la base de datos en una prueba no serán vistos por otra. Por ejemplo, ambas de las siguientes pruebas pasarán:

```
from django.test import TestCase
from myapp.models import Thing

class MyTest(TestCase):

    def test_1(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create()
        self.assertEqual(Thing.objects.count(), 1)

    def test_2(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create(attr1="value")
        self.assertEqual(Thing.objects.count(), 1)
```

Accesorios

Si desea que los objetos de la base de datos utilicen varias pruebas, puede crearlos en el método de `setUp` del caso de prueba. Además, si ha definido los accesorios en su proyecto de django, se pueden incluir así:

```
class MyTest(TestCase):
    fixtures = ["fixture1.json", "fixture2.json"]
```

Por defecto, django está buscando accesorios en el directorio de `fixtures` en cada aplicación. Se pueden configurar otros directorios utilizando la configuración `FIXTURE_DIRS` :

```
# myapp/settings.py
FIXTURE_DIRS = [
    os.path.join(BASE_DIR, 'path', 'to', 'directory'),
]
```

Supongamos que ha creado un modelo de la siguiente manera:

```
# models.py
from django.db import models


class Person(models.Model):
    """A person defined by his/her first- and lastname."""
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
```

Entonces tus accesorios .json podrían verse así:

```
# fixture1.json
[
    { "model": "myapp.person",
        "pk": 1,
        "fields": {
            "firstname": "Peter",
            "lastname": "Griffin"
        }
    },
    { "model": "myapp.person",
        "pk": 2,
        "fields": {
            "firstname": "Louis",
            "lastname": "Griffin"
        }
    },
]
```

Reutilizar la base de datos de prueba.

Para acelerar las pruebas de ejecución, puede indicar al comando de administración que reutilice la base de datos de prueba (y que evite que se cree antes y se elimine después de cada ejecución de prueba). Esto se puede hacer usando la marca keepdb (o taquigrafía `-k`) de esta manera:

```
# Reuse the test-database (since django version 1.8)
$ python manage.py test --keepdb
```

Limitar el número de pruebas ejecutadas.

Es posible limitar las pruebas ejecutadas por `manage.py test` especificando qué módulos debe descubrir el corredor de prueba:

```
# Run only tests for the app names "app1"
$ python manage.py test app1

# If you split the tests file into a module with several tests files for an app
$ python manage.py test app1.tests.test_models

# it's possible to dig down to individual test methods.
$ python manage.py test app1.tests.test_models.MyTestCase.test_something
```

Si desea ejecutar un montón de pruebas, puede pasar un patrón de nombres de archivos. Por ejemplo, es posible que desee ejecutar solo pruebas que involucren a sus modelos:

```
$ python manage.py test -p test_models*
Creating test database for alias 'default'...
..............................................
----------------------------------------------------------------------
Ran 115 tests in 3.869s

OK
```

Finalmente, es posible detener el conjunto de pruebas en el primer fallo, utilizando `--failfast` . Este argumento permite obtener rápidamente el error potencial encontrado en la suite:

```
$ python manage.py test app1
...F..
----------------------------------------------------------------------
Ran 6 tests in 0.977s

FAILED (failures=1)


$ python manage.py test app1 --failfast
...F
======================================================================
[Traceback of the failing test]
----------------------------------------------------------------------
Ran 4 tests in 0.372s

FAILED (failures=1)
```

Lea Examen de la unidad en línea: https://riptutorial.com/es/django/topic/1232/examen-de-la-unidad

Capítulo 23: Explotación florestal

Examples

Iniciar sesión en el servicio Syslog

Es posible configurar Django para generar un registro de salida a un servicio de syslog local o remoto. Esta configuración utiliza el SysLogHandler incorporado en python .

```
from logging.handlers import SysLogHandler
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format' : "[YOUR PROJECT NAME] [%(asctime)s] %(levelname)s [%(name)s:%(lineno)s]
%(message)s",
            'datefmt' : "%d/%b/%Y %H:%M:%S"
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
        'syslog': {
            'class': 'logging.handlers.SysLogHandler',
            'formatter': 'standard',
            'facility': 'user',
            # uncomment next line if rsyslog works with unix socket only (UDP reception
disabled)
            #'address': '/dev/log'
        }
    },
    'loggers': {
        'django':{
            'handlers': ['syslog'],
            'level': 'INFO',
            'disabled': False,
            'propagate': True
        }
    }
}

# loggers for my apps, uses INSTALLED_APPS in settings
# each app must have a configured logger
# level can be changed as desired: DEBUG, INFO, WARNING...
MY_LOGGERS = {}
for app in INSTALLED_APPS:
    MY_LOGGERS[app] = {
        'handlers': ['syslog'],
        'level': 'DEBUG',
        'propagate': True,
    }
LOGGING['loggers'].update(MY_LOGGERS)
```

Configuración básica de registro de Django.

Internamente, Django usa el sistema de registro Python. Hay muchas formas de configurar el registro de un proyecto. Aquí hay una base:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': "[%(asctime)s] %(levelname)s [%(name)s:%(lineno)s] %(message)s",
            'datefmt': "%Y-%m-%d %H:%M:%S"
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'default'
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
            'level': 'INFO',
        },
    }
}
```

Formateadores

Se puede usar para configurar la aparición de los registros cuando se imprimen en la salida. Puede definir muchos formateadores configurando una cadena de clave para cada formateador diferente. Luego se usa un formateador cuando se declara un controlador.

Manipuladores

Se puede utilizar para configurar dónde se imprimirán los registros. En el ejemplo anterior, se envían a stdout y stderr. Hay varias clases de manejador:

```
'rotated_logs': {
    'class': 'logging.handlers.RotatingFileHandler',
    'filename': '/var/log/my_project.log',
    'maxBytes': 1024 * 1024 * 5,  # 5 MB
    'backupCount': 5,
    'formatter': 'default'
    'level': 'DEBUG',
},
```

Esto producirá registros en el archivo seleccionado por `filename` de `filename` . En este ejemplo, se creará un nuevo archivo de registro cuando el actual alcance el tamaño de 5 MB (el anterior se renombrará a my_project.log.1) y los últimos 5 archivos se guardarán para archivar.

```
'mail_admins': {
    'level': 'ERROR',
    'class': 'django.utils.log.AdminEmailHandler'
},
```

Esto enviará cada registro por eamil a los usuarios especificados en la variable de configuración `ADMINS` . El nivel se establece en `ERROR` , por lo que solo los registros con nivel `ERROR` se enviarán por correo electrónico. Esto es extremadamente útil para mantenerse informado sobre posibles errores 50x en un servidor de producción.

Otros manejadores pueden usarse con Django. Para una lista completa, por favor lea la documentación correspondiente. Al igual que los formateadores, puede definir muchos manejadores en un mismo proyecto, estableciendo para cada cadena de clave diferente. Cada controlador se puede utilizar en un registrador específico.

Madereros

En `LOGGING` , la última parte configura para cada módulo el nivel de registro mínimo, los controladores a usar, etc.

Lea Explotación florestal en línea: https://riptutorial.com/es/django/topic/1231/explotacion-florestal

Capítulo 24: Extendiendo o Sustituyendo Modelo de Usuario

Examples

Modelo de usuario personalizado con correo electrónico como campo de inicio de sesión principal.

modelos.py:

```
from __future__ import unicode_literals
from django.db import models
from django.contrib.auth.models import (
        AbstractBaseUser, BaseUserManager, PermissionsMixin)
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _


class UserManager(BaseUserManager):
    def _create_user(self, email,password, is_staff, is_superuser, **extra_fields):
        now = timezone.now()
        if not email:
            raise ValueError('users must have an email address')
        email = self.normalize_email(email)
        user = self.model(email = email,
                          is_staff = is_staff,
                          is_superuser = is_superuser,
                          last_login = now,
                          date_joined = now,
                          **extra_fields)
        user.set_password(password)
        user.save(using = self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        user = self._create_user(email, password, False, False, **extra_fields)
        return user

    def create_superuser(self, email, password, **extra_fields):
        user = self._create_user(email, password, True, True, **extra_fields)
        return user

class User(AbstractBaseUser,PermissionsMixin):
    """My own custom user class"""

    email = models.EmailField(max_length=255, unique=True, db_index=True,
verbose_name=_('email address'))
    date_joined = models.DateTimeField(auto_now_add=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = 'email'
```

```
    REQUIRED_FIELDS = []

    class Meta:
        verbose_name = _('user')
        verbose_name_plural = _('users')

    def get_full_name(self):
    """Return the email."""
        return self.email

    def get_short_name(self):
    """Return the email."""
        return self.email
```

forms.py:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from .models import User


class RegistrationForm(UserCreationForm):
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email'}),
        label="Email")
    password1 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':'password1'}),
        label="Password")
    password2 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name': 'password2'}),
        label="Password (again)")

    '''added attributes so as to customise for styling, like bootstrap'''
    class Meta:
        model = User
        fields = ['email','password1','password2']
        field_order = ['email','password1','password2']

    def clean(self):
    """
    Verifies that the values entered into the password fields match
    NOTE : errors here will appear in 'non_field_errors()'
    """
        cleaned_data = super(RegistrationForm, self).clean()
        if 'password1' in self.cleaned_data and 'password2' in self.cleaned_data:
            if self.cleaned_data['password1'] != self.cleaned_data['password2']:
                raise forms.ValidationError("Passwords don't match. Please try again!")
        return self.cleaned_data

    def save(self, commit=True):
        user = super(RegistrationForm,self).save(commit=False)
        user.set_password(self.cleaned_data['password1'])
        if commit:
            user.save()
        return user

#The save(commit=False) tells Django to save the new record, but dont commit it to the
database yet

class AuthenticationForm(forms.Form): # Note: forms.Form NOT forms.ModelForm
```

```
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email','placeholder':'Email'}),
        label='Email')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':
'password','placeholder':'Password'}),
        label='Password')

    class Meta:
        fields = ['email', 'password']
```

views.py:

```
from django.shortcuts import redirect, render, HttpResponse
from django.contrib.auth import login as django_login, logout as django_logout, authenticate
as django_authenticate
#importing as such so that it doesn't create a confusion with our methods and django's default
methods

from django.contrib.auth.decorators import login_required
from .forms import AuthenticationForm, RegistrationForm


def login(request):
    if request.method == 'POST':
        form = AuthenticationForm(data = request.POST)
        if form.is_valid():
            email = request.POST['email']
            password = request.POST['password']
            user = django_authenticate(email=email, password=password)
            if user is not None:
                if user.is_active:
                    django_login(request,user)
                    return redirect('/dashboard') #user is redirected to dashboard
    else:
        form = AuthenticationForm()

    return render(request,'login.html',{'form':form,})

def register(request):
    if request.method == 'POST':
        form = RegistrationForm(data = request.POST)
        if form.is_valid():
            user = form.save()
            u = django_authenticate(user.email = user, user.password = password)
            django_login(request,u)
            return redirect('/dashboard')
    else:
        form = RegistrationForm()

    return render(request,'register.html',{'form':form,})

def logout(request):
    django_logout(request)
    return redirect('/')

@login_required(login_url ="/")
def dashboard(request):
    return render(request, 'dashboard.html',{})
```

settings.py:

```
AUTH_USER_MODEL = 'myapp.User'
```

admin.py

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import Group
from .models import User


class UserAdmin(BaseUserAdmin):
    list_display = ('email','is_staff')
    list_filter = ('is_staff',)
    fieldsets = ((None,
                  {'fields':('email','password')}), ('Permissions',{'fields':('is_staff',)}),)
    add_fieldsets = ((None, {'classes': ('wide',), 'fields': ('email', 'password1',
'password2')}),)
    search_fields =('email',)
    ordering = ('email',)
    filter_horizontal = ()

admin.site.register(User, UserAdmin)
admin.site.unregister(Group)
```

Usa el `email` como nombre de usuario y deshazte del campo` username`

Si desea deshacerse del campo de `username` de `username` y utilizar el `email` como identificador único de usuario, tendrá que crear un modelo de `User` personalizado que extienda `AbstractBaseUser` lugar de `AbstractUser` . De hecho, el `username` y el `email` se definen en `AbstractUser` y no puede anularlos. Esto significa que también tendrá que redefinir todos los campos que desee que estén definidos en `AbstractUser` .

```
from django.contrib.auth.models import (
    AbstractBaseUser, PermissionsMixin, BaseUserManager,
)
from django.db import models
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _

class UserManager(BaseUserManager):

    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        if not email:
            raise ValueError('The given email must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', False)
```

```
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

    return self._create_user(email, password, **extra_fields)


class User(AbstractBaseUser, PermissionsMixin):
    """PermissionsMixin contains the following fields:
        - `is_superuser`
        - `groups`
        - `user_permissions`
     You can omit this mix-in if you don't want to use permissions or
     if you want to implement your own permissions logic.
     """

    class Meta:
        verbose_name = _("user")
        verbose_name_plural = _("users")
        db_table = 'auth_user'
        # `db_table` is only needed if you move from the existing default
        # User model to a custom one. This enables to keep the existing data.

    USERNAME_FIELD = 'email'
    """Use the email as unique username."""

    REQUIRED_FIELDS = ['first_name', 'last_name']

    GENDER_MALE = 'M'
    GENDER_FEMALE = 'F'
    GENDER_CHOICES = [
        (GENDER_MALE, _("Male")),
        (GENDER_FEMALE, _("Female")),
    ]

    email = models.EmailField(
        verbose_name=_("email address"), unique=True,
        error_messages={
            'unique': _(
                "A user is already registered with this email address"),
        },
    )
    gender = models.CharField(
        max_length=1, blank=True, choices=GENDER_CHOICES,
        verbose_name=_("gender"),
    )
    first_name = models.CharField(
        max_length=30, verbose_name=_("first name"),
    )
    last_name = models.CharField(
        max_length=30, verbose_name=_("last name"),
    )
    is_staff = models.BooleanField(
```

```
        verbose_name=_("staff status"),
        default=False,
        help_text=_(
            "Designates whether the user can log into this admin site."
        ),
    )
    is_active = models.BooleanField(
        verbose_name=_("active"),
        default=True,
        help_text=_(
            "Designates whether this user should be treated as active. "
            "Unselect this instead of deleting accounts."
        ),
    )
    date_joined = models.DateTimeField(
        verbose_name=_("date joined"), default=timezone.now,
    )

    objects = UserManager()
```

Amplíe el modelo de usuario de Django fácilmente

Nuestra clase `UserProfile`

Cree una clase de modelo `UserProfile` con la relación de `OneToOne` con el modelo de `User` predeterminado:

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    photo = FileField(verbose_name=_("Profile Picture"),
                      upload_to=upload_to("main.UserProfile.photo", "profiles"),
                      format="Image", max_length=255, null=True, blank=True)
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)
    phone = models.CharField(max_length=20, blank=True, default='')
    city = models.CharField(max_length=100, default='', blank=True)
    country = models.CharField(max_length=100, default='', blank=True)
    organization = models.CharField(max_length=100, default='', blank=True)
```

Django Signals en el trabajo

Usando Django Signals, cree un nuevo `UserProfile` `User` inmediatamente después de crear un objeto de `User` . Esta función se puede colocar debajo de la clase de modelo `UserProfile` en el mismo archivo, o colocarla donde desee. No me importa, siempre y cuando lo refiera correctamente.

```
def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

`inlineformset_factory` al rescate

Ahora para tus `views.py` , podrías tener algo como esto:

```
from django.shortcuts import render, HttpResponseRedirect
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from .models import UserProfile
from .forms import UserForm
from django.forms.models import inlineformset_factory
from django.core.exceptions import PermissionDenied
@login_required() # only logged in users should access this
def edit_user(request, pk):
    # querying the User object with pk from url
    user = User.objects.get(pk=pk)

    # prepopulate UserProfileForm with retrieved user values from above.
    user_form = UserForm(instance=user)

    # The sorcery begins from here, see explanation https://blog.khophi.co/extending-django-
user-model-userprofile-like-a-pro/
    ProfileInlineFormset = inlineformset_factory(User, UserProfile, fields=('website', 'bio',
'phone', 'city', 'country', 'organization'))
    formset = ProfileInlineFormset(instance=user)

    if request.user.is_authenticated() and request.user.id == user.id:
        if request.method == "POST":
            user_form = UserForm(request.POST, request.FILES, instance=user)
            formset = ProfileInlineFormset(request.POST, request.FILES, instance=user)

            if user_form.is_valid():
                created_user = user_form.save(commit=False)
                formset = ProfileInlineFormset(request.POST, request.FILES,
instance=created_user)

                if formset.is_valid():
                    created_user.save()
                    formset.save()
                    return HttpResponseRedirect('/accounts/profile/')

        return render(request, "account/account_update.html", {
            "noodle": pk,
            "noodle_form": user_form,
            "formset": formset,
        })
    else:
        raise PermissionDenied
```

Nuestra plantilla

Luego, escupe todo a tu plantilla `account_update.html` así:

```
{% load material_form %}
<!-- Material form is just a materialize thing for django forms -->
<div class="col s12 m8 offset-m2">
      <div class="card">
        <div class="card-content">
        <h2 class="flow-text">Update your information</h2>
          <form action="." method="POST" class="padding">
```

```
            {% csrf_token %} {{ noodle_form.as_p }}
            <div class="divider"></div>
            {{ formset.management_form }}
                {{ formset.as_p }}
            <button type="submit" class="btn-floating btn-large waves-light waves-effect"><i
class="large material-icons">done</i></button>
            <a href="#" onclick="window.history.back(); return false;" title="Cancel"
class="btn-floating waves-effect waves-light red"><i class="material-icons">history</i></a>

        </form>
        </div>
    </div>
</div>
```

Sobre el fragmento de código tomado de la extensión del perfil de usuario de Django como un profesional

Especificación de un modelo de usuario personalizado

El modelo de `User` incorporado de Django no siempre es apropiado para algunos tipos de proyectos. En algunos sitios, podría tener más sentido utilizar una dirección de correo electrónico en lugar de un nombre de usuario, por ejemplo.

Puede anular el modelo de `User` predeterminado al agregar su modelo de `User` personalizado a la configuración `AUTH_USER_MODEL` , en el archivo de configuración de sus proyectos:

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

Tenga en cuenta que es altamente `AUTH_USER_MODEL` crear `AUTH_USER_MODEL` antes de crear cualquier migración o ejecutar `manage.py migrate` por primera vez. Debido a las limitaciones de la característica de dependencia synamic de Django.

Por ejemplo, en su blog, es posible que desee que otros autores puedan iniciar sesión con una dirección de correo electrónico en lugar del nombre de usuario habitual, por lo que creamos un modelo de `User` personalizado con una dirección de correo electrónico como `USERNAME_FIELD` :

```
from django.contrib.auth.models import AbstractBaseUser

class CustomUser(AbstractBaseUser):
     email = models.EmailField(unique=True)

     USERNAME_FIELD = 'email'
```

Al heredar el `AbstractBaseUser` podemos construir un modelo de `User` compatible. `AbstractBaseUser` proporciona la implementación central de un modelo de `User` .

Para que el comando Django `manage.py createsuperuser` sepa qué otros campos son necesarios, podemos especificar un `REQUIRED_FIELDS` . Este valor no tiene efecto en otras partes de Django!

```
class CustomUser(AbstractBaseUser):
    ...
    first_name = models.CharField(max_length=254)
```

```
    last_name = models.CharField(max_length=254)
    ...
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

Para cumplir con otra parte de Django, todavía tenemos que especificar el valor `is_active` , las funciones `get_full_name()` y `get_short_name()` :

```
class CustomUser(AbstractBaseUser):
    ...
    is_active = models.BooleanField(default=False)
    ...
    def get_full_name(self):
        full_name = "{0} {1}".format(self.first_name, self.last_name)
        return full_name.strip()

    def get_short_name(self):
        return self.first_name
```

También debe crear un `UserManager` personalizado para su modelo de `User` , que le permita a Django usar las `create_user()` y `create_superuser()` :

```
from django.contrib.auth.models import BaseUserManager

class CustomUserManager(BaseUserManager):
    def create_user(self, email, first_name, last_name, password=None):
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
        )

        user.set_password(password)
        user.first_name = first_name
        user.last_name = last_name
        user.save(using=self._db)
        return user

    def create_superuser(self, email, first_name, last_name, password):
        user = self.create_user(
            email=email,
            first_name=first_name,
            last_name=last_name,
            password=password,
        )

        user.is_admin = True
        user.is_active = True
        user.save(using=self.db)
        return user
```

Referencia al modelo de usuario

Su código no funcionará en proyectos en los que haga referencia al modelo de `User` (*y donde se haya cambiado la configuración de* `AUTH_USER_MODEL`) directamente.

Por ejemplo: si desea crear `Post` modelo de `Post` para un blog con un modelo de `User` personalizado, debe especificar el modelo de `User` personalizado de esta manera:

```
from django.conf import settings
from django.db import models

class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
```

Lea Extendiendo o Sustituyendo Modelo de Usuario en línea:
https://riptutorial.com/es/django/topic/1209/extendiendo-o-sustituyendo-modelo-de-usuario

Capítulo 25: F () expresiones

Introducción

Una expresión F () es una forma en que Django puede usar un objeto de Python para referirse al valor del campo del modelo o la columna anotada en la base de datos sin tener que extraer el valor en la memoria de Python. Esto permite a los desarrolladores evitar ciertas condiciones de carrera y también filtrar los resultados según los valores de campo del modelo.

Sintaxis

- desde django.db.models import F

Examples

Evitando las condiciones de carrera.

Vea esta pregunta de preguntas y respuestas si no sabe cuáles son las condiciones de la raza.

El siguiente código puede estar sujeto a condiciones de carrera:

```
article = Article.objects.get(pk=69)
article.views_count += 1
article.save()
```

Si `views_count` es igual a `1337` , esto resultará en dicha consulta:

```
UPDATE app_article SET views_count = 1338 WHERE id=69
```

Si dos clientes acceden a este artículo al mismo tiempo, lo que *puede* suceder es que la segunda solicitud HTTP ejecute `Article.objects.get(pk=69)` antes de que el primero ejecute `article.save()` . Por lo tanto, ambas solicitudes tendrán `views_count = 1337` , lo incrementarán y guardarán `views_count = 1338` en la base de datos, mientras que en realidad debería ser `1339` .

Para arreglar esto, usa una expresión `F()` :

```
article = Article.objects.get(pk=69)
article.views_count = F('views_count') + 1
article.save()
```

Esto, por otro lado, resultará en dicha consulta:

```
UPDATE app_article SET views_count = views_count + 1 WHERE id=69
```

Actualizando queryset a granel

Supongamos que queremos eliminar 2 upvotes de todos los artículos del autor con id `51` .
Hacer esto solo con Python ejecutaría `N` consultas (siendo `N` la cantidad de artículos en el conjunto de consultas):

```
for article in Article.objects.filter(author_id=51):
    article.upvotes -= 2
    article.save()
    # Note that there is a race condition here but this is not the focus
    # of this example.
```

¿Qué pasaría si en lugar de juntar todos los artículos en Python, hacer un bucle sobre ellos, disminuir los votos positivos y guardar cada uno actualizado en la base de datos, hubiera otra manera?
Usando una expresión `F()` , puede hacerlo en una consulta:

```
Article.objects.filter(author_id=51).update(upvotes=F('upvotes') - 2)
```

Que se puede traducir en la siguiente consulta SQL:

```
UPDATE app_article SET upvotes = upvotes - 2 WHERE author_id = 51
```

¿Por qué es esto mejor?

- En lugar de Python haciendo el trabajo, pasamos la carga a la base de datos que se ajusta para realizar dichas consultas.
- Reduce efectivamente el número de consultas de base de datos necesarias para lograr el resultado deseado.

Ejecutar operaciones aritméticas entre campos.

`F()` expresiones `F()` se pueden usar para ejecutar operaciones aritméticas (`+` , `-` , `*` etc.) entre los campos del modelo, para definir una búsqueda / conexión algebraica entre ellos.

- Que el modelo sea:

  ```
  class MyModel(models.Model):
      int_1 = models.IntegerField()
      int_2 = models.IntegerField()
  ```

- Ahora supongamos que queremos recuperar todos los objetos de la tabla `MyModel` `int_2` campos `int_1` e `int_2` satisfacen esta ecuación: `int_1 + int_2 >= 5` . Utilizando `annotate()` y `filter()` obtenemos:

  ```
  result = MyModel.objects.annotate(
              diff=F(int_1) + F(int_2)
          ).filter(diff__gte=5)
  ```

  `result` ahora contiene todos los objetos antes mencionados.

Aunque el ejemplo utiliza campos `Integer` , este método funcionará en todos los campos en los que se pueda aplicar una operación aritmética.

Lea F () expresiones en línea: https://riptutorial.com/es/django/topic/2765/f----expresiones

Capítulo 26: filtro django

Examples

Utilice django-filter con CBV

`django-filter` es un sistema genérico para filtrar Django QuerySets según las selecciones de los usuarios. La documentación lo utiliza en una vista basada en funciones como modelo de producto:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField()
    description = models.TextField()
    release_date = models.DateField()
    manufacturer = models.ForeignKey(Manufacturer)
```

El filtro será el siguiente:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr='iexact')

    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

Para usar esto en un CBV, anule `get_queryset()` de ListView, luego devuelva el conjunto de `querset` filtrado:

```
from django.views.generic import ListView
from .filters import ProductFilter

class ArticleListView(ListView):
    model = Product

    def get_queryset(self):
        qs = self.model.objects.all()
        product_filtered_list = ProductFilter(self.request.GET, queryset=qs)
        return product_filtered_list.qs
```

Es posible acceder a los objetos filtrados en sus vistas, como con la paginación, en `f.qs` Esto paginará la lista de objetos filtrados.

Lea filtro django en línea: https://riptutorial.com/es/django/topic/6101/filtro-django

Capítulo 27: Form Widgets

Examples

Widget de entrada de texto simple

El ejemplo más simple de widget es el ingreso de texto personalizado. Por ejemplo, para crear un `<input type="tel">` , debe subclase `TextInput` y establecer `input_type` en `'tel'` .

```
from django.forms.widgets import TextInput

class PhoneInput(TextInput):
    input_type = 'tel'
```

Widget compuesto

Puede crear widgets compuestos de múltiples widgets utilizando `MultiWidget` .

```
from datetime import date

from django.forms.widgets import MultiWidget, Select
from django.utils.dates import MONTHS

class SelectMonthDateWidget(MultiWidget):
    """This widget allows the user to fill in a month and a year.

    This represents the first day of this month or, if `last_day=True`, the
    last day of this month.
    """

    default_nb_years = 10

    def __init__(self, attrs=None, years=None, months=None, last_day=False):
        self.last_day = last_day

        if not years:
            this_year = date.today().year
            years = range(this_year, this_year + self.default_nb_years)
        if not months:
            months = MONTHS

        # Here we will use two `Select` widgets, one for months and one for years
        widgets = (Select(attrs=attrs, choices=months.items()),
                   Select(attrs=attrs, choices=((y, y) for y in years)))
        super().__init__(widgets, attrs)

    def format_output(self, rendered_widgets):
        """Concatenates rendered sub-widgets as HTML"""
        return (
            '<div class="row">'
            '<div class="col-xs-6">{}</div>'
            '<div class="col-xs-6">{}</div>'
            '</div>'
        ).format(*rendered_widgets)
```

```
    def decompress(self, value):
        """Split the widget value into subwidgets values.
        We expect value to be a valid date formated as `%Y-%m-%d`.
        We extract month and year parts from this string.
        """
        if value:
            value = date(*map(int, value.split('-')))
            return [value.month, value.year]
        return [None, None]

    def value_from_datadict(self, data, files, name):
        """Get the value according to provided `data` (often from `request.POST`)
        and `files` (often from `request.FILES`, not used here)
        `name` is the name of the form field.

        As this is a composite widget, we will grab multiple keys from `data`.
        Namely: `field_name_0` (the month) and `field_name_1` (the year).
        """
        datalist = [
            widget.value_from_datadict(data, files, '{}_{}'.format(name, i))
            for i, widget in enumerate(self.widgets)]
        try:
            # Try to convert it as the first day of a month.
            d = date(day=1, month=int(datelist[0]), year=int(datelist[1]))
            if self.last_day:
                # Transform it to the last day of the month if needed
                if d.month == 12:
                    d = d.replace(day=31)
                else:
                    d = d.replace(month=d.month+1) - timedelta(days=1)
        except (ValueError, TypeError):
            # If we failed to recognize a valid date
            return ''
        else:
            # Convert it back to a string with format `%Y-%m-%d`
            return str(d)
```

Lea Form Widgets en línea: https://riptutorial.com/es/django/topic/1230/form-widgets

Capítulo 28: Formas

Examples

Ejemplo de ModelForm

Cree un ModelForm a partir de una clase Model existente, subclasificando `ModelForm` :

```
from django import forms

class OrderForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['item', 'order_date', 'customer', 'status']
```

Definiendo un formulario Django desde cero (con widgets)

Los formularios se pueden definir, de manera similar a los modelos, subclasificando `django.forms.Form` .
Varias opciones de entrada de campo están disponibles, como `CharField` , `URLField` , `IntegerField` , etc.

La definición de un formulario de contacto simple se puede ver a continuación:

```
from django import forms

class ContactForm(forms.Form):
    contact_name = forms.CharField(
        label="Your name", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    contact_email = forms.EmailField(
        label="Your Email Address", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    content = forms.CharField(
        label="Your Message", required=True,
        widget=forms.Textarea(attrs={'class': 'form-control'}))
```

Widget es la representación de Django de las etiquetas de entrada de usuario HTML y se puede usar para representar html personalizados para campos de formulario (por ejemplo, como se representa un cuadro de texto para la entrada de contenido aquí)

`attrs` atributos son atributos que se copiarán tal como están en el HTML procesado para el formulario.

Por ejemplo: `content.render("name", "Your Name")` da

```
<input title="Your name" type="text" name="name" value="Your Name" class="form-control" />
```

Eliminando un campo de modelForm basado en la condición de views.py

Si tenemos un modelo como el siguiente,

```
from django.db import models
from django.contrib.auth.models import User

class UserModuleProfile(models.Model):
    user = models.OneToOneField(User)
    expired = models.DateTimeField()
    admin = models.BooleanField(default=False)
    employee_id = models.CharField(max_length=50)
    organisation_name = models.ForeignKey('Organizations', on_delete=models.PROTECT)
    country = models.CharField(max_length=100)
    position = models.CharField(max_length=100)

    def __str__(self):
        return self.user
```

Y una forma modelo que utiliza este modelo como sigue,

```
from .models import UserModuleProfile, from django.contrib.auth.models import User
from django import forms

class UserProfileForm(forms.ModelForm):
    admin = forms.BooleanField(label="Make this User
Admin",widget=forms.CheckboxInput(),required=False)
    employee_id = forms.CharField(label="Employee Id ")
    organisation_name = forms.ModelChoiceField(label='Organisation
Name',required=True,queryset=Organizations.objects.all(),empty_label="Select an Organization")
    country = forms.CharField(label="Country")
    position = forms.CharField(label="Position")

    class Meta:
        model = UserModuleProfile
        fields = ('admin','employee_id','organisation_name','country','position',)

    def __init__(self, *args, **kwargs):
        admin_check = kwargs.pop('admin_check', False)
        super(UserProfileForm, self).__init__(*args, **kwargs)
        if not admin_check:
            del self.fields['admin']
```

Tenga en cuenta que debajo de la clase Meta en el formulario agregué una función de **inicio** que podemos usar mientras inicializamos el formulario desde views.py para eliminar un campo de formulario (o algunas otras acciones). Te lo explicaré más tarde.

Por lo tanto, este formulario se puede utilizar para fines de registro de usuario y queremos que todos los campos estén definidos en la clase Meta del formulario. Pero, ¿qué pasa si queremos usar el mismo formulario cuando editamos al usuario pero cuando lo hacemos no queremos mostrar el campo de administración del formulario?

Simplemente podemos enviar un argumento adicional cuando inicializamos el formulario en base a alguna lógica y eliminamos el campo de administración del backend.

```
def edit_profile(request,user_id):
    context = RequestContext(request)
```

```
    user = get_object_or_404(User, id=user_id)
    profile = get_object_or_404(UserModuleProfile, user_id=user_id)
    admin_check = False
    if request.user.is_superuser:
        admin_check = True
    # If it's a HTTP POST, we're interested in processing form data.
    if request.method == 'POST':
        # Attempt to grab information from the raw form information.
        profile_form =
UserProfileForm(data=request.POST,instance=profile,admin_check=admin_check)
        # If the form is valid...
        if profile_form.is_valid():
            form_bool = request.POST.get("admin", "xxx")
            if form_bool == "xxx":
                form_bool_value = False
            else:
                form_bool_value = True
            profile = profile_form.save(commit=False)
            profile.user = user
            profile.admin = form_bool_value
            profile.save()
            edited = True
        else:
            print profile_form.errors

    # Not a HTTP POST, so we render our form using ModelForm instance.
    # These forms will be blank, ready for user input.
    else:
        profile_form = UserProfileForm(instance = profile,admin_check=admin_check)

    return render_to_response(
            'usermodule/edit_user.html',
            {'id':user_id, 'profile_form': profile_form, 'edited': edited, 'user':user},
            context)
```

Como puede ver, he mostrado aquí un ejemplo de edición simple utilizando el formulario que creamos anteriormente. Observe que cuando inicialicé el formulario pasé una variable `admin_check` adicional que contiene `True` o `False` .

```
profile_form = UserProfileForm(instance = profile,admin_check=admin_check)
```

Ahora, si observa el formulario que escribimos anteriormente, puede ver que en el **inicio** intentamos capturar el `admin_check` que pasamos desde aquí. Si el valor es Falso, simplemente eliminamos el Campo de `admin` del formulario y lo usamos. Y dado que este es un modelo, el campo de administrador no puede ser nulo en el modelo, simplemente verificamos si la publicación de formulario tenía un campo de administrador en la publicación de formulario, si no lo configuramos como `False` en el código de vista en el siguiente código de vista.

```
form_bool = request.POST.get("admin", "xxx")
if form_bool == "xxx":
    form_bool_value = False
else:
    form_bool_value = True
```

Cargas de archivos con Django Forms

En primer lugar, debemos agregar `MEDIA_ROOT` y `MEDIA_URL` a nuestro archivo `settings.py`

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

También aquí trabajará con `ImageField` , así que recuerde que en tales casos instale la biblioteca Pillow (`pip install pillow`). De lo contrario, tendrá tal error:

```
ImportError: No module named PIL
```

Pillow es una bifurcación de PIL, la biblioteca de imágenes de Python, que ya no se mantiene. La almohada es compatible con PIL.

Django viene con dos campos de formulario para cargar archivos al servidor, `FileField` e `ImageField` , el siguiente es un ejemplo del uso de estos dos campos en nuestro formulario

forms.py:

```
from django import forms


class UploadDocumentForm(forms.Form):
    file = forms.FileField()
    image = forms.ImageField()
```

views.py:

```
from django.shortcuts import render
from .forms import UploadDocumentForm


def upload_doc(request):
    form = UploadDocumentForm()
    if request.method == 'POST':
        form = UploadDocumentForm(request.POST, request.FILES)  # Do not forget to add:
request.FILES
        if form.is_valid():
            # Do something with our files or simply save them
            # if saved, our files would be located in media/ folder under the project's base
folder
            form.save()
    return render(request, 'upload_doc.html', locals())
```

upload_doc.html:

```
<html>
    <head>File Uploads</head>
    <body>
        <form enctype="multipart/form-data" action="" method="post"> <!-- Do not forget to
add: enctype="multipart/form-data" -->
            {% csrf_token %}
            {{ form }}
            <input type="submit" value="Save">
        </form>
```

```
    </body>
</html>
```

Validación de campos y Confirmar modelo (Cambiar correo electrónico de usuario)

Ya hay formularios implementados dentro de Django para cambiar la contraseña del usuario, un ejemplo es SetPasswordForm .

Sin embargo, no hay formularios para modificar el correo electrónico del usuario y creo que el siguiente ejemplo es importante para entender cómo usar un formulario correctamente.

El siguiente ejemplo realiza las siguientes comprobaciones:

- De hecho, el correo electrónico ha cambiado; es muy útil si necesita validar el correo electrónico o actualizar el correo de chimpancé;
- Tanto el correo electrónico como el correo electrónico de confirmación son los mismos: el formulario tiene dos campos para correo electrónico, por lo que la actualización es menos propensa a errores.

Y al final, guarda el nuevo correo electrónico en el objeto de usuario (actualiza el correo electrónico del usuario). Observe que el `__init__()` requiere un objeto de usuario.

```
class EmailChangeForm(forms.Form):
    """
    A form that lets a user change set their email while checking for a change in the
    e-mail.
    """
    error_messages = {
        'email_mismatch': _("The two email addresses fields didn't match."),
        'not_changed': _("The email address is the same as the one already defined."),
    }

    new_email1 = forms.EmailField(
        label=_("New email address"),
        widget=forms.EmailInput,
    )

    new_email2 = forms.EmailField(
        label=_("New email address confirmation"),
        widget=forms.EmailInput,
    )

    def __init__(self, user, *args, **kwargs):
        self.user = user
        super(EmailChangeForm, self).__init__(*args, **kwargs)

    def clean_new_email1(self):
        old_email = self.user.email
        new_email1 = self.cleaned_data.get('new_email1')
        if new_email1 and old_email:
            if new_email1 == old_email:
                raise forms.ValidationError(
                    self.error_messages['not_changed'],
                    code='not_changed',
```

```
                )
        return new_email1

    def clean_new_email2(self):
        new_email1 = self.cleaned_data.get('new_email1')
        new_email2 = self.cleaned_data.get('new_email2')
        if new_email1 and new_email2:
            if new_email1 != new_email2:
                raise forms.ValidationError(
                    self.error_messages['email_mismatch'],
                    code='email_mismatch',
                )
        return new_email2

    def save(self, commit=True):
        email = self.cleaned_data["new_email1"]
        self.user.email = email
        if commit:
            self.user.save()
        return self.user



def email_change(request):
    form = EmailChangeForm()
    if request.method=='POST':
        form = Email_Change_Form(user,request.POST)
        if form.is_valid():
            if request.user.is_authenticated:
                if form.cleaned_data['email1']  == form.cleaned_data['email2']:
                    user = request.user
                    u = User.objects.get(username=user)
                    # get the proper user
                    u.email = form.cleaned_data['email1']
                    u.save()
                    return HttpResponseRedirect("/accounts/profile/")
    else:
        return render_to_response("email_change.html", {'form':form},
                                  context_instance=RequestContext(request))
```

Lea Formas en línea: https://riptutorial.com/es/django/topic/1217/formas

Capítulo 29: Formsets

Sintaxis

- NewFormSet = formset_factory (SomeForm, extra = 2)
- formset = NewFormSet (initial = [{'some_field': 'Field Value', 'other_field': 'Other Field Value',}])

Examples

Formsets con datos inicializados y unificados.

`Formset` es una forma de representar múltiples formularios en una página, como una cuadrícula de datos. Ej .: Este `ChoiceForm` podría estar asociado con alguna pregunta de ordenación. Como, los niños son los más inteligentes entre qué edad ?.

`appname/forms.py`

```
from django import forms
class ChoiceForm(forms.Form):
    choice = forms.CharField()
    pub_date = forms.DateField()
```

En sus vistas, puede usar el constructor `formset_factory` que toma `Form` como parámetro su `ChoiceForm` en este caso y `extra` que describe cuántas formas adicionales distintas de las formas inicializadas deben procesarse, y puede recorrer el objeto `formset` como cualquier otro. otro iterable.

Si el formset no se inicializa con datos, imprime el número de formularios igual a `extra + 1` y si el formset se inicializa se imprime `initialized + extra` donde hay `extra` número `extra` de formularios vacíos distintos de los inicializados.

`appname/views.py`

```
import datetime
from django.forms import formset_factory
from appname.forms import ChoiceForm
    ChoiceFormSet = formset_factory(ChoiceForm, extra=2)
    formset = ChoiceFormSet(initial=[
      {'choice': 'Between 5-15 ?',
        'pub_date': datetime.date.today(),}
      ])
```

si hace un bucle sobre el `formset object` como este para form in formset: print (form.as_table ())

`Output in rendered template`

```
<tr>
<th><label for="id_form-0-choice">Choice:</label></th>
```

```
<td><input type="text" name="form-0-choice" value="Between 5-15 ?" id="id_form-0-choice"
/></td>
</tr>
<tr>
<th><label for="id_form-0-pub_date">Pub date:</label></th>
<td><input type="text" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date"
/></td>
</tr>
<tr>
<th><label for="id_form-1-choice">Choice:</label></th>
<td><input type="text" name="form-1-choice" id="id_form-1-choice" /></td>
</tr>
<tr>
<th><label for="id_form-1-pub_date">Pub date:</label></th>
<td><input type="text" name="form-1-pub_date" id="id_form-1-pub_date" /></td
</tr>
<tr>
<th><label for="id_form-2-choice">Choice:</label></th>
<td><input type="text" name="form-2-choice" id="id_form-2-choice" /></td>
</tr>
<tr>
<th><label for="id_form-2-pub_date">Pub date:</label></th>
<td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td>
</tr>
```

Lea Formsets en línea: https://riptutorial.com/es/django/topic/6082/formsets

Capítulo 30: Gestores personalizados y Querysets

Examples

Definiendo un administrador básico usando Querysets y el método `as_manager`

Django manger es una interfaz a través de la cual el modelo de django consulta la base de datos. El campo de `objects` utilizado en la mayoría de las consultas de django es en realidad el administrador predeterminado creado por nosotros por django (esto solo se crea si no definimos administradores personalizados).

¿Por qué definiríamos un gestor / queryset personalizado?

Para evitar escribir consultas comunes en todo el código base y, en su lugar, recomendarlas utilizando una abstracción más fácil de recordar. Ejemplo: Decida usted mismo qué versión es más legible:

- Solo obtenga todos los usuarios activos: `User.objects.filter(is_active=True)` vs `User.manager.active()`
- Obtenga todos los dermatólogos activos en nuestro formulario: `User.objects.filter(is_active=True).filter(is_doctor=True).filter(specialization='Dermatology')` vs `User.manager.doctors.with_specialization('Dermatology')`

Otro beneficio es que si mañana decidimos que todos los `psychologists` también son `dermatologists` , podemos modificar fácilmente la consulta en nuestro Gerente y terminar con ella.

A continuación se muestra un ejemplo de creación de un `Manager` personalizado definido mediante la creación de un `QuerySet` y el uso del método `as_manager` .

```
from django.db.models.query import QuerySet

class ProfileQuerySet(QuerySet):
    def doctors(self):
        return self.filter(user_type="Doctor", user__is_active=True)

    def with_specializations(self, specialization):
        return self.filter(specializations=specialization)

    def users(self):
        return self.filter(user_type="Customer", user__is_active=True)

ProfileManager = ProfileQuerySet.as_manager
```

Lo añadiremos a nuestro modelo de la siguiente manera:

```
class Profile(models.Model):
```

```
    ...
    manager = ProfileManager()
```

NOTA : Una vez que hayamos definido un `manager` en nuestro modelo, los `objects` ya no se definirán para el modelo.

select_related para todas las consultas

Modelo con ForeignKey

Trabajaremos con estos modelos:

```
from django.db import models

class Book(models.Model):
 name= models.CharField(max_length=50)
 author = models.ForeignKey(Author)

class Author(models.Model):
 name = models.CharField(max_length=50)
```

Supongamos que a menudo (siempre) accedemos a `book.author.name`

En vista

Podríamos usar lo siguiente, cada vez,

```
books = Book.objects.select_related('author').all()
```

Pero esto no es SECO.

Gestor personalizado

```
class BookManager(models.Manager):

    def get_queryset(self):
        qs = super().get_queryset()
        return qs.select_related('author')

class Book(models.Model):
    ...
    objects = BookManager()
```

Nota : la llamada a `super` debe cambiarse para python 2.x

Ahora todo lo que tenemos que usar en vistas es

```
books = Book.objects.all()
```

y no se harán consultas adicionales en la plantilla / vista.

Definir gestores personalizados.

Muy a menudo sucede que se trata de modelos que tienen algo así como un campo `published` . Este tipo de campos se utilizan casi siempre al recuperar objetos, por lo que se encontrará a sí mismo escribiendo algo como:

```
my_news = News.objects.filter(published=True)
```

demasiadas veces. Puedes usar administradores personalizados para lidiar con estas situaciones, de modo que luego puedas escribir algo como:

```
my_news = News.objects.published()
```

que es más agradable y más fácil de leer por otros desarrolladores también.

Cree un archivo `managers.py` en el directorio de su aplicación y defina una nueva clase de `models.Manager` :

```
from django.db import models


class NewsManager(models.Manager):

    def published(self, **kwargs):
        # the method accepts **kwargs, so that it is possible to filter
        # published news
        # i.e: News.objects.published(insertion_date__gte=datetime.now)
        return self.filter(published=True, **kwargs)
```

use esta clase redefiniendo la propiedad de `objects` en la clase modelo:

```
from django.db import models

# import the created manager
from .managers import NewsManager

class News(models.Model):
    """ News model
    """
    insertion_date = models.DateTimeField('insertion date', auto_now_add=True)
    title = models.CharField('title', max_length=255)
    # some other fields here
    published = models.BooleanField('published')

    # assign the manager class to the objects property
    objects = NewsManager()
```

Ahora puedes obtener tus noticias publicadas simplemente de esta manera:

```
my_news = News.objects.published()
```

y también puedes realizar más filtrado:

```
my_news = News.objects.published(title__icontains='meow')
```

Lea Gestores personalizados y Querysets en línea:
https://riptutorial.com/es/django/topic/1400/gestores-personalizados-y-querysets

Capítulo 31: Integración continua con Jenkins

Examples

Jenkins 2.0+ Pipeline Script

Las versiones modernas de Jenkins (versión 2.x) vienen con un "Build Pipeline Plugin" que se puede usar para organizar tareas de CI complejas sin crear una multitud de trabajos interconectados, y le permite controlar fácilmente la configuración de compilación / compilación.

Puede instalarlo manualmente en un trabajo de tipo "Pipeline" o, si su proyecto está alojado en Github, puede usar el "Complemento de la carpeta de la organización de GitHub" para configurar trabajos automáticamente para usted.

Aquí hay una configuración simple para los sitios de Django que requieren que solo se instalen los módulos de python especificados del sitio.

```
#!/usr/bin/groovy

node {
  // If you are having issues with your project not getting updated,
  // try uncommenting the following lines.
  //stage 'Checkout'
  //checkout scm
  //sh 'git submodule update --init --recursive'

  stage 'Update Python Modules'
  // Create a virtualenv in this folder, and install or upgrade packages
  // specified in requirements.txt; https://pip.readthedocs.io/en/1.1/requirements.html
  sh 'virtualenv env && source env/bin/activate && pip install --upgrade -r requirements.txt'

  stage 'Test'
  // Invoke Django's tests
  sh 'source env/bin/activate && python ./manage.py runtests'
}
```

Jenkins 2.0+ Pipeline Script, Docker Containers

Aquí hay un ejemplo de un script de canalización que construye un contenedor Docker y luego ejecuta las pruebas dentro de él. El punto de entrada se supone que es ya sea `manage.py` o `invoke` / `fabric` con una `runtests` de comandos disponibles.

```
#!/usr/bin/groovy

node {
  stage 'Checkout'
  checkout scm
  sh 'git submodule update --init --recursive'
```

```
  imageName = 'mycontainer:build'
  remotes = [
    'dockerhub-account',
  ]

  stage 'Build'
  def djangoImage = docker.build imageName

  stage 'Run Tests'
  djangoImage.run('', 'runtests')

  stage 'Push'
  for (int i = 0; i < remotes.size(); i++) {
      sh "docker tag ${imageName} ${remotes[i]}/${imageName}"
      sh "docker push ${remotes[i]}/${imageName}"
   }
}
```

Lea Integración continua con Jenkins en línea:
https://riptutorial.com/es/django/topic/5873/integracion-continua-con-jenkins

Capítulo 32: Internacionalización

Sintaxis

- gettext (mensaje)
- ngettext (singular, plural, número)
- ugettext (mensaje)
- ungettext (singular, plural, número)
- pgettext (contexto, mensaje)
- npgettext (contexto, singular, plural, número)
- gettext_lazy (mensaje)
- ngettext_lazy (singular, plural, número = ninguno)
- ugettext_lazy (mensaje)
- ungettext_lazy (singular, plural, número = ninguno)
- pgettext_lazy (contexto, mensaje)
- npgettext_lazy (contexto, singular, plural, número = ninguno)
- gettext_noop (mensaje)
- ugettext_noop (mensaje)

Examples

Introducción a la internacionalización

Configurando

settings.py

```
from django.utils.translation import ugettext_lazy as _

USE_I18N = True  # Enable Internationalization
LANGUAGE_CODE = 'en'  # Language in which original texts are written
LANGUAGES = [  # Available languages
    ('en', _("English")),
    ('de', _("German")),
    ('fr', _("French")),
]

# Make sure the LocaleMiddleware is included, AFTER SessionMiddleware
# and BEFORE middlewares using internationalization (such as CommonMiddleware)
MIDDLEWARE_CLASSES = [
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
]
```

Marcando cuerdas como traducibles

El primer paso en la traducción es *marcar cadenas como traducibles* . Esto es pasarlos a través de una de las funciones `gettext` (vea la sección de Sintaxis). Por ejemplo, aquí hay un ejemplo de definición de modelo:

```
from django.utils.translation import ugettext_lazy as _
# It is common to import gettext as the shortcut `_` as it is often used
# several times in the same file.

class Child(models.Model):

    class Meta:
        verbose_name = _("child")
        verbose_name_plural = _("children")

    first_name = models.CharField(max_length=30, verbose_name=_("first name"))
    last_name = models.CharField(max_length=30, verbose_name=_("last name"))
    age = models.PositiveSmallIntegerField(verbose_name=_("age"))
```

Todas las cadenas encapsuladas en `_()` ahora están marcadas como traducibles. Cuando se imprimen, siempre se mostrarán como la cadena encapsulada, independientemente del idioma elegido (ya que todavía no hay traducción disponible).

Traduciendo cuerdas

Este ejemplo es suficiente para comenzar con la traducción. La mayoría de las veces, solo querrá marcar las cadenas como traducibles para **anticipar la posible internacionalización** de su proyecto. Por lo tanto, esto se cubre en otro ejemplo .

Lazy vs Non-Lazy traducción

Cuando se usa una traducción no perezosa, las cadenas se traducen inmediatamente.

```
>>> from django.utils.translation import activate, ugettext as _
>>> month = _("June")
>>> month
'June'
>>> activate('fr')
>>> _("June")
'juin'
>>> activate('de')
>>> _("June")
'Juni'
>>> month
'June'
```

Cuando se usa la pereza, la traducción solo ocurre cuando realmente se usa.

```
>>> from django.utils.translation import activate, ugettext_lazy as _
```

```
>>> month = _("June")
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> str(month)
'June'
>>> activate('fr')
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> "month: {}".format(month)
'month: juin'
>>> "month: %s" % month
'month: Juni'
```

Tienes que usar la traducción perezosa en los casos en que:

- La traducción puede no estar activada (idioma no seleccionado) cuando se evalúa `_("some string")`
- Algunas cadenas pueden evaluarse solo al inicio (por ejemplo, en atributos de clase tales como definiciones de campos de formulario y modelo)

Traducción en plantillas

Para habilitar la traducción en plantillas, debe cargar la biblioteca `i18n` .

```
{% load i18n %}
```

La traducción básica se realiza con la etiqueta `trans` template.

```
{% trans "Some translatable text" %}
{# equivalent to python `ugettext("Some translatable text")` #}
```

La etiqueta `trans` template soporta el contexto:

```
{% trans "May" context "month" %}
{# equivalent to python `pgettext("May", "month")` #}
```

Para incluir marcadores de posición en su cadena de traducción, como en:

```
_("My name is {first_name} {last_name}").format(first_name="John", last_name="Doe")
```

Tendrás que usar la etiqueta de la plantilla `blocktrans` :

```
{% blocktrans with first_name="John" last_name="Doe" %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Por supuesto, en lugar de `"John"` y `"Doe"` , puedes tener variables y filtros:

```
{% blocktrans with first_name=user.first_name last_name=user.last_name|title %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Si `first_name` y `last_name` ya están en su contexto, incluso puede omitir la cláusula `with` :

```
{% blocktrans %}My name is {{ first_name }} {{ last_name }}{% endblocktrans %}
```

Sin embargo, solo se pueden usar variables de contexto de "nivel superior". **Esto no funcionará:**

```
{% blocktrans %}
    My name is {{ user.first_name }} {{ user.last_name }}
{% endblocktrans %}
```

Esto se debe principalmente a que el nombre de la variable se utiliza como marcador de posición en los archivos de traducción.

La etiqueta de la plantilla `blocktrans` también acepta la pluralización.

```
{% blocktrans count nb=users|length }}
    There is {{ nb }} user.
{% plural %}
    There are {{ nb }} users.
{% endblocktrans %}
```

Finalmente, independientemente de la biblioteca `i18n` , puede pasar cadenas traducibles a etiquetas de plantilla usando la sintaxis `_("")` .

```
{{ site_name|default:_("It works!") }}
{% firstof var1 var2 _("translatable fallback") %}
```

Este es un sistema de plantillas django incorporado mágico para imitar una sintaxis de llamada de función, pero no es una llamada de función. `_("It works!")` Pasado a la etiqueta de plantilla `default` como una cadena `'_("It works!")'` Que luego se analiza como una cadena traducible, tal como el `name` se analiza como una variable y el `"name"` es analizado como una cadena.

Traduciendo cuerdas

Para traducir cadenas, deberás crear archivos de traducción. Para hacerlo, django se envía con los comandos de gestión `makemessages` .

```
$ django-admin makemessages -l fr
processing locale fr
```

El comando anterior descubrirá todas las cadenas marcadas como traducibles dentro de las aplicaciones instaladas y creará un archivo de idioma para cada aplicación para la traducción al francés. Por ejemplo, si solo tiene una aplicación `myapp` contiene cadenas traducibles, esto creará un archivo `myapp/locale/fr/LC_MESSAGES/django.po` . Este archivo puede parecerse a lo siguiente:

```
# SOME DESCRIPTIVE TITLE
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR
```

```
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-07-24 14:01+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: myapp/models.py:22
msgid "user"
msgstr ""

#: myapp/models.py:39
msgid "A user already exists with this email address."
msgstr ""

#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
```

Primero tendrá que rellenar los marcadores de posición (resaltados con mayúsculas). Luego traduce las cuerdas. `msgid` es la cadena marcada como traducible en su código. `msgstr` es donde tienes que escribir la traducción de la cadena de arriba.

Cuando una cadena contiene marcadores de posición, también deberá incluirlos en su traducción. Por ejemplo, traducirás el último mensaje de la siguiente manera:

```
#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
"En vous inscrivant, vous acceptez nos <a href=\"%(terms_url)s\" "
"target=_blank>Conditions d'utilisation</a>"
```

Una vez que se termina su archivo de traducción, tendrá que compilar los `.po` archivos en `.mo` archivos. Esto se hace llamando al `compilemessages` gestión de `compilemessages` :

```
$ django-admin compilemessages
```

Eso es todo, ahora las traducciones están disponibles.

Para actualizar sus archivos de traducción cuando realice cambios en su código, puede volver a ejecutar `django-admin makemessages -l fr` . Esto actualizará los archivos `.po` , manteniendo sus

traducciones existentes y agregando las nuevas. Las cadenas eliminadas todavía estarán disponibles en los comentarios. Para actualizar archivos `.po` para todos los idiomas, ejecute `django-admin makemessages -a` . Una vez que se `.po` archivos `.po` , no olvide ejecutar nuevamente los `django-admin compilemessages` para generar archivos `.mo` .

Caso de uso noop

`(u)gettext_noop` permite marcar una cadena como traducible sin traducirla realmente.

Un caso de uso típico es cuando desea registrar un mensaje para desarrolladores (en inglés) pero también desea mostrarlo al cliente (en el idioma solicitado). **Puede pasar una variable a `gettext` , pero su contenido no se descubrirá como una cadena traducible porque es, por definición, variable.** .

```
# THIS WILL NOT WORK AS EXPECTED
import logging
from django.contrib import messages

logger = logging.getLogger(__name__)

error_message = "Oops, something went wrong!"
logger.error(error_message)
messages.error(request, _(error_message))
```

El mensaje de error no aparecerá en el archivo `.po` y deberá recordar que existe para agregarlo manualmente. Para solucionar esto, puede utilizar `gettext_noop` .

```
error_message = ugettext_noop("Oops, something went wrong!")
logger.error(error_message)
messages.error(request, _(error_message))
```

Ahora la cadena `"Oops, something went wrong!"` se descubrirá y estará disponible en el archivo `.po` cuando se genere. Y el error todavía se registrará en inglés para los desarrolladores.

Errores comunes

traducciones difusas

A veces, los `makemessages` pueden pensar que la cadena que encontró para la traducción es algo similar a la traducción ya existente. Lo hará cuando lo marque en el archivo `.po` con un comentario `fuzzy` especial como este:

```
#: templates/randa/map.html:91
#, fuzzy
msgid "Country"
msgstr "Länderinfo"
```

Incluso si la traducción es correcta o si la actualizaste para corregir una, no se utilizará para traducir tu proyecto a menos que elimines `fuzzy` línea de comentarios `fuzzy` .

Cuerdas multilínea

`makemessages` analiza los archivos en varios formatos, desde texto plano a código Python y no está diseñado para seguir todas las reglas posibles para tener cadenas de varias líneas en esos formatos. La mayoría de las veces funcionará bien con cadenas de una sola línea, pero si tiene una construcción como esta:

```
translation = _("firstline"
"secondline"
"thirdline")
```

Solo recogerá `firstline` para la traducción. La solución para esto es evitar el uso de cadenas multilínea cuando sea posible.

Lea Internacionalización en línea: https://riptutorial.com/es/django/topic/2579/internacionalizacion

Capítulo 33: JSONField - un campo específico de PostgreSQL

Sintaxis

- JSONField (** opciones)

Observaciones

- El `JSONField` de Django realmente almacena los datos en una columna `JSONB` Postgres, que solo está disponible en Postgres 9.4 y posteriores.
- `JSONField` es genial cuando quieres un esquema más flexible. Por ejemplo, si desea cambiar las claves sin tener que realizar ninguna migración de datos, o si no todos sus objetos tienen la misma estructura.
- Si está almacenando datos con claves estáticas, considere usar varios campos normales en lugar de `JSONField` s, ya que la consulta a `JSONField` puede ser bastante tediosa a veces.

Encadenar consultas

Puedes encadenar consultas juntas. Por ejemplo, si un diccionario existe dentro de una lista, agregue dos guiones bajos y su consulta de diccionario.

> No te olvides de separar las consultas con guiones bajos.

Examples

Creando un campo JSON

Disponible en Django 1.9+

```
from django.contrib.postgres.fields import JSONField
from django.db import models

class IceCream(models.Model):
    metadata = JSONField()
```

Puede agregar las `**options` normales `**options` si lo desea.

> ***!*** Tenga en cuenta que debe poner `'django.contrib.postgres'` en `INSTALLED_APPS` en su `settings.py`

Creando un objeto con datos en un campo JSON

Pase datos en forma nativa de Python, por ejemplo, `list` , `dict` , `str` , `None` , `bool` , etc.

```
IceCream.objects.create(metadata={
    'date': '1/1/2016',
    'ordered by': 'Jon Skeet',
    'buyer': {
         'favorite flavor': 'vanilla',
         'known for': ['his rep on SO', 'writing a book']
     },
    'special requests': ['hot sauce'],
})
```

Consulte la nota en la sección "Comentarios" sobre el uso de `JSONField` en la práctica.

Consulta de datos de nivel superior.

```
IceCream.objects.filter(metadata__ordered_by='Guido Van Rossum')
```

Consulta de datos anidados en diccionarios.

Obtenga todos los conos de helado que fueron ordenados por personas que les gusta el chocolate:

```
IceCream.objects.filter(metadata__buyer__favorite_flavor='chocolate')
```

Consulte la nota en la sección "Comentarios" sobre el encadenamiento de consultas.

Consulta de datos presentes en matrices

Un entero se interpretará como una búsqueda de índice.

```
IceCream.objects.filter(metadata__buyer__known_for__0='creating stack overflow')
```

Consulte la nota en la sección "Comentarios" sobre el encadenamiento de consultas.

Ordenar por valores JSONField

El pedido directamente en `JSONField` todavía no se admite en Django. Pero es posible a través de RawSQL usando las funciones PostgreSQL para jsonb:

```
from django.db.models.expressions import RawSQL
RatebookDataEntry.objects.all().order_by(RawSQL("data->>%s", ("json_objects_key",)))
```

Este ejemplo ordena por `data['json_objects_key']` dentro de `JSONField` nombre de `data` :

```
data = JSONField()
```

Lea JSONField - un campo específico de PostgreSQL en línea:
https://riptutorial.com/es/django/topic/1759/jsonfield---un-campo-especifico-de-postgresql

Capítulo 34: Mapeo de cadenas a cadenas con HStoreField - un campo específico de PostgreSQL

Sintaxis

- FooModel.objects.filter (field_name__key_name = 'valor a consultar')

Examples

Configurando HStoreField

Primero, tendremos que hacer alguna configuración para que `HStoreField` funcione.

1. asegúrese de que `django.contrib.postgres` esté en su `INSTALLED_APPS
2. Agregue `HStoreExtension` a sus migraciones. Recuerde poner `HStoreExtension` antes de cualquier migración de `CreateModel` o `AddField` .

```
from django.contrib.postgres.operations import HStoreExtension
from django.db import migrations

class FooMigration(migrations.Migration):
    # put your other migration stuff here
    operations = [
        HStoreExtension(),
        ...
    ]
```

Agregando HStoreField a tu modelo

> -> Nota: asegúrese de configurar `HStoreField` primero antes de continuar con este ejemplo. (encima)

No se requieren parámetros para inicializar un `HStoreField` .

```
from django.contrib.postgres.fields import HStoreField
from django.db import models

class Catalog(models.model):
    name = models.CharField(max_length=200)
    titles_to_authors = HStoreField()
```

Creando una nueva instancia de modelo

Pase un diccionario de python nativo asignando cadenas a cadenas para `create()` .

```
Catalog.objects.create(name='Library of Congress', titles_to_authors={
    'Using HStoreField with Django': 'CrazyPython and la communidad',
    'Flabbergeists and thingamajigs': 'La Artista Fooista',
    'Pro Git': 'Scott Chacon and Ben Straub',
})
```

Realizar búsquedas de claves

```
Catalog.objects.filter(titles__Pro_Git='Scott Chacon and Ben Straub')
```

Usando contiene

Pase un objeto `dict` a `field_name__contains` como un argumento de palabra clave.

```
Catalog.objects.filter(titles__contains={
        'Pro Git': 'Scott Chacon and Ben Straub'})
```

Equivalente al operador de SQL `@>`.

Lea Mapeo de cadenas a cadenas con HStoreField - un campo específico de PostgreSQL en línea: https://riptutorial.com/es/django/topic/2670/mapeo-de-cadenas-a-cadenas-con-hstorefield---un-campo-especifico-de-postgresql

Capítulo 35: Meta: Pautas de documentación.

Observaciones

Esta es una extensión de "Meta: Pautas de documentación" de Python para Django.

Estas son solo propuestas, no recomendaciones. Siéntase libre de editar cualquier cosa aquí si no está de acuerdo o si tiene algo más que mencionar.

Examples

Las versiones no compatibles no necesitan una mención especial

Es poco probable que alguien use una versión no compatible de Django, y bajo su propio riesgo. Si alguna vez alguien lo hace, debe preocuparle saber si existe una característica en la versión dada.

Teniendo en cuenta lo anterior, es inútil mencionar las especificidades de una versión no compatible.

1.6

Este tipo de bloque es inútil porque ninguna persona sana usa Django <1.6.

1.8

Este tipo de bloque es inútil porque ninguna persona sana usa Django <1.8.

Esto también se aplica a los temas. En el momento de escribir este ejemplo, las vistas de clase basadas en estados admitidos son versiones `1.3-1.9` . Podemos asumir con seguridad que esto es en realidad equivalente a `All versions` . Esto también evita actualizar todos los temas de las versiones compatibles cada vez que se lanza una nueva versión.

> Las versiones actuales soportadas son: `1.8` [1] `1.9` [2] `1.10` [1]

1. Correcciones de seguridad, errores de pérdida de datos, errores de bloqueo, errores de funcionalidad principales en las características recién introducidas y regresiones de versiones anteriores de Django.
2. Correcciones de seguridad y errores de pérdida de datos.

Lea Meta: Pautas de documentación. en línea: https://riptutorial.com/es/django/topic/5243/meta--pautas-de-documentacion-

Capítulo 36: Middleware

Introducción

Middleware en Django es un marco que permite que el código se enganche en el procesamiento de respuesta / solicitud y altere la entrada o salida de Django.

Observaciones

Es necesario agregar middleware a su configuración.py `MIDDLEWARE_CLASSES` lista antes de que se incluya en la ejecución. La lista predeterminada que proporciona Django al crear un nuevo proyecto es la siguiente:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Estas son todas las funciones que se ejecutarán en **orden** en cada solicitud (una vez antes de que alcance su código de vista en `views.py` y una vez en orden inverso para devolución de llamada `process_response` , antes de la versión 1.10). Hacen una variedad de cosas, como inyectar el token de falsificación de solicitudes en sitios cruzados (csrf) .

El orden importa porque si algún middleware realiza una redirección, el middleware subsiguiente nunca se ejecutará. O si un middleware espera que el token csrf esté allí, tiene que ejecutarse después del `CsrfViewMiddleware` .

Examples

Añadir datos a las solicitudes

Django hace que sea realmente fácil agregar datos adicionales a las solicitudes de uso dentro de la vista. Por ejemplo, podemos analizar el subdominio en el META de la solicitud y adjuntarlo como una propiedad separada en la solicitud mediante el uso de middleware.

```
class SubdomainMiddleware:
    def process_request(self, request):
        """
        Parse out the subdomain from the request
        """
        host = request.META.get('HTTP_HOST', '')
        host_s = host.replace('www.', '').split('.')
```

```
        request.subdomain = None
        if len(host_s) > 2:
            request.subdomain = host_s[0]
```

Si agrega datos con middleware a su solicitud, puede acceder a los datos recién agregados más adelante en la línea. Aquí usaremos el subdominio analizado para determinar algo como qué organización está accediendo a su aplicación. Este enfoque es útil para las aplicaciones que se implementan con una configuración de DNS con subdominios comodín que apuntan a una sola instancia y la persona que accede a la aplicación desea una versión con apariencia que dependa del punto de acceso.

```
class OrganizationMiddleware:
    def process_request(self, request):
        """
        Determine the organization based on the subdomain
        """
        try:
            request.org = Organization.objects.get(domain=request.subdomain)
        except Organization.DoesNotExist:
            request.org = None
```

Recuerde que el orden es importante cuando el middleware depende del otro. Para las solicitudes, deseará que el middleware dependiente se coloque después de la dependencia.

```
MIDDLEWARE_CLASSES = [
    ...
    'myapp.middleware.SubdomainMiddleware',
    'myapp.middleware.OrganizationMiddleware',
    ...
]
```

Middleware para filtrar por dirección IP

Primero: La estructura del camino.

Si no lo tiene, debe crear la carpeta de **middleware** dentro de su aplicación siguiendo la estructura:

```
yourproject/yourapp/middleware
```

La carpeta de middleware se debe colocar en la misma carpeta que settings.py, urls, templates ...

Importante: no olvide crear el archivo init .py vacío dentro de la carpeta de middleware para que su aplicación reconozca esta carpeta

En lugar de tener una carpeta separada que contenga sus clases de middleware, también es posible poner sus funciones en un solo archivo, `yourproject/yourapp/middleware.py` .

Segundo: Crea el middleware.

Ahora debemos crear un archivo para nuestro middleware personalizado. En este ejemplo, supongamos que queremos un middleware que filtre a los usuarios según su dirección IP, creamos un archivo llamado **filter_ip_middleware.py** :

```
#yourproject/yourapp/middleware/filter_ip_middleware.py
from django.core.exceptions import PermissionDenied

class FilterIPMiddleware(object):
    # Check if client IP address is allowed
    def process_request(self, request):
        allowed_ips = ['192.168.1.1', '123.123.123.123', etc...] # Authorized ip's
        ip = request.META.get('REMOTE_ADDR') # Get client IP address
        if ip not in allowed_ips:
            raise PermissionDenied # If user is not allowed raise Error

       # If IP address is allowed we don't do anything
       return None
```

Tercero: Agregue el middleware en nuestro 'settings.py'

Necesitamos buscar `MIDDLEWARE_CLASSES` dentro de settings.py y allí debemos agregar nuestro middleware (agregarlo *en la última posición*). Debería ser como

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
     # Above are Django standard middlewares

     # Now we add here our custom middleware
     'yourapp.middleware.filter_ip_middleware.FilterIPMiddleware'
)
```

¡Hecho! Ahora, cada solicitud de cada cliente llamará a su middleware personalizado y procesará su código personalizado.

Excepción de manejo global

Supongamos que ha implementado alguna lógica para detectar intentos de modificar un objeto en la base de datos, mientras que el cliente que envió los cambios no tuvo las últimas modificaciones. Si tal caso ocurre, `ConfictError(detailed_message)` una excepción personalizada `ConfictError(detailed_message)` .

Ahora desea devolver un código de estado HTTP 409 (Conficto) cuando se produce este error. Por lo general, puede usarlo como middleware para esto en lugar de manejarlo en cada vista que pueda generar esta excepción.

```
class ConfictErrorHandlingMiddleware:
    def process_exception(self, request, exception):
        if not isinstance(exception, ConflictError):
            return  # Propagate other exceptions, we only handle ConflictError
```

```
        context = dict(confict_details=str(exception))
        return TemplateResponse(request, '409.html', context, status=409)
```

Entendiendo el nuevo estilo del middleware Django 1.10

Django 1.10 introdujo un nuevo estilo de middleware donde se combinan `process_request` y `process_response` .

En este nuevo estilo, *un middleware es un invocable que devuelve otro invocable* . Bueno, en realidad la **primera es una fábrica de middleware** y **la última es el middleware real** .

La *fábrica de middleware* toma como único argumento el siguiente *middleware* en la pila de middlewares, o la vista en sí misma cuando se alcanza la parte inferior de la pila.

El *middleware* toma la solicitud como único argumento y **siempre devuelve un `HttpResponse` .**

El mejor ejemplo para ilustrar cómo funciona el *middleware de* nuevo estilo es probablemente mostrar cómo hacer un *middleware* compatible con versiones anteriores:

```
class MyMiddleware:

    def __init__(self, next_layer=None):
        """We allow next_layer to be None because old-style middlewares
        won't accept any argument.
        """
        self.get_response = next_layer

    def process_request(self, request):
        """Let's handle old-style request processing here, as usual."""
        # Do something with request
        # Probably return None
        # Or return an HttpResponse in some cases

    def process_response(self, request, response):
        """Let's handle old-style response processing here, as usual."""
        # Do something with response, possibly using request.
        return response

    def __call__(self, request):
        """Handle new-style middleware here."""
        response = self.process_request(request)
        if response is None:
            # If process_request returned None, we must call the next middleware or
            # the view. Note that here, we are sure that self.get_response is not
            # None because this method is executed only in new-style middlewares.
            response = self.get_response(request)
        response = self.process_response(request, response)
        return response
```

Lea Middleware en línea: https://riptutorial.com/es/django/topic/1721/middleware

Capítulo 37: Migraciones

Parámetros

comando `django-admin`	Detalles
`makemigrations <my_app>`	Generar migraciones para `my_app`
`makemigrations`	Generar migraciones para todas las aplicaciones.
`makemigrations --merge`	Resolver conflictos de migración para todas las aplicaciones.
`makemigrations --merge <my_app>`	Resolver conflictos de migración para `my_app`
`makemigrations --name <migration_name> <my_app>`	Genera una migración para `my_app` con el nombre `migration_name`
`migrate <my_app>`	Aplicar migraciones pendientes de `my_app` a la base de datos.
`migrate`	Aplicar todas las migraciones pendientes a la base de datos.
`migrate <my_app> <migration_name>`	Aplicar o no aplicar hasta el nombre de `migration_name`
`migrate <my_app> zero`	Desplegar todas las migraciones en `my_app`
`sqlmigrate <my_app> <migration_name>`	Imprime el SQL para la migración nombrada.
`showmigrations`	Muestra todas las migraciones para todas las aplicaciones.
`showmigrations <my_app>`	Muestra todas las migraciones en `my_app`

Examples

Trabajando con migraciones

Django usa migraciones para propagar los cambios que realiza a sus modelos en su base de datos. La mayoría de las veces, django puede generarlos por ti.

Para crear una migración, ejecute:

```
$ django-admin makemigrations <app_name>
```

Esto creará un archivo de migración en el submódulo de `migration` de `app_name` . La primera migración se llamará `0001_initial.py` , la otra comenzará con `0002_` , luego `0003` , ...

Si omite `<app_name>` esto creará migraciones para todas sus `INSTALLED_APPS` .

Para propagar migraciones a su base de datos, ejecute:

```
$ django-admin migrate <app_name>
```

Para mostrar todas sus migraciones, ejecute:

```
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [X] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

- `[X]` significa que la migración se propagó a su base de datos
- `[ ]` significa que la migración no se propagó a su base de datos. Usa `django-admin migrate` para propagarlo

También puede llamar a revertir migraciones, esto se puede hacer pasando el nombre de la `migrate command` . Dada la lista anterior de migraciones (mostrada por `django-admin showmigrations`):

```
$ django-admin migrate app_name 0002  # Roll back to migration 0002
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [ ] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

Migraciones manuales

A veces, las migraciones generadas por Django no son suficientes. Esto es especialmente cierto cuando desea realizar **migraciones de datos** .

Por ejemplo, vamos a tener ese modelo:

```
class Article(models.Model):
    title = models.CharField(max_length=70)
```

Este modelo ya tiene datos existentes y ahora desea agregar un `SlugField` :

```
class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70)
```

Usted creó las migraciones para agregar el campo, pero ahora le gustaría establecer la babosa

para todos los artículos existentes, de acuerdo con su `title` .

Por supuesto, puedes hacer algo como esto en la terminal:

```
$ django-admin shell
>>> from my_app.models import Article
>>> from django.utils.text import slugify
>>> for article in Article.objects.all():
...     article.slug = slugify(article.title)
...     article.save()
...
>>>
```

Pero tendrá que hacer esto en todos sus entornos (es decir, en el escritorio de su oficina, en su computadora portátil, ...), todos sus compañeros de trabajo también deberán hacerlo, y tendrán que pensar en ello durante la puesta en escena y al momento de presionar. vivir.

Para hacerlo de una vez por todas, lo haremos en una migración. Primero crea una migración vacía:

```
$ django-admin makemigrations --empty app_name
```

Esto creará un archivo de migración vacío. Ábrelo, contiene un esqueleto base. Digamos que su migración anterior se llamó `0023_article_slug` y esta se llama `0024_auto_20160719_1734` . Esto es lo que escribiremos en nuestro archivo de migración:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.9.7 on 2016-07-19 15:34
from __future__ import unicode_literals

from django.db import migrations
from django.utils.text import slugify


def gen_slug(apps, schema_editor):
    # We can't import the Article model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Article = apps.get_model('app_name', 'Article')
    for row in Article.objects.all():
        row.slug = slugify(row.name)
        row.save()


class Migration(migrations.Migration):

    dependencies = [
        ('hosting', '0023_article_slug'),
    ]

    operations = [
        migrations.RunPython(gen_slug, reverse_code=migrations.RunPython.noop),
        # We set `reverse_code` to `noop` because we cannot revert the migration
        # to get it back in the previous state.
        # If `reverse_code` is not given, the migration will not be reversible,
        # which is not the behaviour we expect here.
    ]
```

Migraciones falsas

Cuando se ejecuta una migración, Django almacena el nombre de la migración en una tabla django_migrations.

Crear y falsificar las migraciones iniciales para el esquema existente.

Si su aplicación ya tiene modelos y tablas de base de datos, y no tiene migraciones. Primero crea migraciones iniciales para tu aplicación.

```
python manage.py makemigrations your_app_label
```

Ahora falsas migraciones iniciales según lo aplicado

```
python manage.py migrate --fake-initial
```

Falsas todas las migraciones en todas las aplicaciones.

```
python manage.py migrate --fake
```

Falsas migraciones de una sola aplicación.

```
python manage.py migrate --fake core
```

Fake único archivo de migración

```
python manage.py migrate myapp migration_name
```

Nombres personalizados para archivos de migración

Use la `makemigrations --name <your_migration_name>` para permitir nombrar las migraciones en lugar de usar un nombre generado.

```
python manage.py makemigrations --name <your_migration_name> <app_name>
```

Resolviendo conflictos migratorios.

Introducción

A veces, las migraciones entran en conflicto, lo que hace que la migración no tenga éxito. Esto puede suceder en muchos escenarios, sin embargo, puede ocurrir de manera regular al desarrollar una aplicación con un equipo.

Los conflictos de migración comunes ocurren mientras se usa el control de origen, especialmente cuando se usa el método de característica por rama. Para este escenario, usaremos un modelo llamado `Reporter` con el `name` y la `address` los atributos.

Dos desarrolladores en este punto van a desarrollar una característica, por lo que ambos obtienen esta copia inicial del modelo `Reporter` . El desarrollador A agrega una `age` que da como resultado el archivo `0002_reporter_age.py` . El desarrollador B agrega un campo `bank_account` que da como resultado `0002_reporter_bank_account` . Una vez que estos desarrolladores fusionan su código e intentan migrar las migraciones, se produjo un conflicto de migración.

Este conflicto se produce porque estas migraciones alteran el mismo modelo, `Reporter` . Además de eso, los nuevos archivos comienzan con 0002.

Fusionando migraciones

Hay varias maneras de hacerlo. Lo siguiente está en el orden recomendado:

1. La solución más simple para esto es ejecutar el comando makemigrations con una marca --merge.

   ```
   python manage.py makemigrations --merge <my_app>
   ```

   Esto creará una nueva migración para resolver el conflicto anterior.

2. Cuando este archivo adicional no es bienvenido en el entorno de desarrollo por razones personales, una opción es eliminar las migraciones en conflicto. Luego, se puede hacer una nueva migración usando el comando regular `makemigrations` . Cuando se escriben migraciones personalizadas, como `migrations.RunPython` , se deben tener en cuenta utilizando este método.

Cambiar un campo de caracteres a una clave foránea

En primer lugar, supongamos que este es su modelo inicial, dentro de una aplicación llamada `discography` :

```
from django.db import models

class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
```

Ahora, te das cuenta de que quieres usar una ForeignKey para el artista. Este es un proceso algo complejo, que se debe realizar en varios pasos.

Paso 1, agregue un nuevo campo para ForeignKey, asegurándose de marcarlo como nulo (tenga en cuenta que el modelo al que estamos vinculando también se incluye ahora):

```
from django.db import models

class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
    artist_link = models.ForeignKey('Artist', null=True)
```

```
class Artist(models.Model):
    name = models.CharField(max_length=255)
```

... y crea una migración para este cambio.

```
./manage.py makemigrations discography
```

Paso 2, rellena tu nuevo campo. Para hacer esto, tienes que crear una migración vacía.

```
./manage.py makemigrations --empty --name transfer_artists discography
```

Una vez que tenga esta migración vacía, desea agregar una sola operación `RunPython` para vincular sus registros. En este caso, podría verse algo como esto:

```
def link_artists(apps, schema_editor):
    Album = apps.get_model('discography', 'Album')
    Artist = apps.get_model('discography', 'Artist')
    for album in Album.objects.all():
        artist, created = Artist.objects.get_or_create(name=album.artist)
        album.artist_link = artist
        album.save()
```

Ahora que sus datos se transfirieron al nuevo campo, podría terminar y dejar todo como está, usando el nuevo campo `artist_link` para todo. O, si desea hacer un poco de limpieza, desea crear dos migraciones más.

Para su primera migración, querrá eliminar su campo original, `artist` . Para su segunda migración, cambie el nombre del nuevo campo `artist_link` a `artist` .

Esto se realiza en varios pasos para garantizar que Django reconozca las operaciones correctamente.

Lea Migraciones en línea: https://riptutorial.com/es/django/topic/1200/migraciones

Capítulo 38: Modelo de referencia de campo

Parámetros

Parámetro	Detalles
nulo	Si es verdadero, los valores vacíos pueden almacenarse como `null` en la base de datos
blanco	Si es verdadero, entonces el campo no será requerido en los formularios. Si los campos se dejan en blanco, Django usará el valor de campo predeterminado.
elecciones	Se puede utilizar un iterable de iterables de 2 elementos como opciones para este campo. Si se establece, el campo se representa como un menú desplegable en el administrador. `[('m', 'Male'),('f','Female'),('z','Prefer Not to Disclose')]` . Para agrupar opciones, simplemente anide los valores: `[('Video Source',((1,'YouTube'),(2,'Facebook')),('Audio Source',((3, 'Soundcloud'),(4, 'Spotify'))]`
db_column	Por defecto, django usa el nombre del campo para la columna de la base de datos. Use esto para proporcionar un nombre personalizado
db_index	Si es `True` , se creará un índice en este campo en la base de datos.
db_tablespace	El espacio de tabla a usar para el índice de este campo. *Este campo solo se usa si el motor de la base de datos lo admite, de lo contrario se ignora* .
defecto	El valor predeterminado para este campo. Puede ser un valor, o un objeto llamable. Para los valores predeterminados mutables (una lista, un conjunto, un diccionario), **debe** utilizar un llamador. Debido a la compatibilidad con las migraciones, no puede utilizar lambdas.
editable	Si es `False` , el campo no se muestra en el administrador del modelo o en cualquier `ModelForm` . El valor predeterminado es `True`
error de mensajes	Se utiliza para personalizar los mensajes de error predeterminados que se muestran para este campo. El valor es un diccionario, con las claves que representan el error y el valor es el mensaje. Las claves predeterminadas (para los mensajes de error) son `null` , en `blank` , `invalid` , `invalid_choice` , `unique` y `unique_for_date` ; los mensajes de error adicionales se pueden definir por campos personalizados.
texto de ayuda	Texto que se mostrará con el campo, para ayudar a los usuarios. HTML está permitido.

Parámetro	Detalles
on_delete	Cuando se elimina un objeto al que hace referencia ForeignKey, Django emula el comportamiento de la restricción de SQL especificada por el argumento on_delete. Este es el segundo argumento posicional para los campos `ForeignKey` y `OneToOneField` . Otros campos no tienen este argumento.
Clave primaria	Si es `True` , este campo será la clave principal. Django agrega automáticamente una clave principal; por lo tanto, esto solo es necesario si desea crear una clave principal personalizada. Sólo puede tener una clave principal por modelo.
único	Si es `True` , se `True` errores si se ingresan valores duplicados para este campo. Esta es una restricción de nivel de base de datos, y no simplemente un bloque de interfaz de usuario.
unique_for_date	Establezca el valor en un `DateField` o `DateTimeField` , y se generarán errores si hay valores duplicados *para la misma fecha o fecha* .
unique_for_month	Similar a `unique_for_date` , excepto que los cheques están limitados para el mes.
unique_for_year	Similar a `unique_for_date` , excepto que los cheques están limitados al año.
verbose_name	Un nombre descriptivo para el campo, utilizado por django en varios lugares (como crear etiquetas en los formularios de administrador y modelo).
validadores	Una lista de validadores para este campo.

Observaciones

- Puede escribir sus propios campos si lo considera necesario.
- Puede anular las funciones de la clase del modelo base, más comúnmente la función `save()`

Examples

Campos de números

Se dan ejemplos de campos numéricos:

AutoField

Un entero auto-incremental generalmente usado para claves primarias.

```
from django.db import models

class MyModel(models.Model):
    pk = models.AutoField()
```

Cada modelo obtiene un campo de clave principal (llamado `id`) de forma predeterminada. Por lo tanto, no es necesario duplicar un campo de identificación en el modelo para los fines de una clave principal.

BigIntegerField

Un número entero de ajuste de `-9223372036854775808` a `9223372036854775807` (`8 Bytes`).

```
from django.db import models

class MyModel(models.Model):
    number_of_seconds = models.BigIntegerField()
```

Campo integral

IntegerField se utiliza para almacenar valores enteros desde -2147483648 hasta 2147483647 (`4 Bytes`).

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.IntegerField(default=0)
```

`default` parámetro por `default` no es obligatorio. Pero es útil establecer un valor predeterminado.

PositiveIntegerField

Como un campo Integer, pero debe ser positivo o cero (0). El PositiveIntegerField se utiliza para almacenar valores enteros de 0 a 2147483647 (`4 Bytes`). Esto puede ser útil en un campo que debería ser semánticamente positivo. Por ejemplo, si está registrando alimentos con sus calorías, no debería ser negativo. Este campo evitará valores negativos a través de sus validaciones.

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.PositiveIntegerField(default=0)
```

`default` parámetro por `default` no es obligatorio. Pero es útil establecer un valor predeterminado.

SmallIntegerField

SmallIntegerField se utiliza para almacenar valores enteros de -32768 a 32767 (`2 Bytes`). Este campo es útil para que los valores no sean extremos.

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=255)
    temperature = models.SmallIntegerField(null=True)
```

PositiveSmallIntegerField

SmallIntegerField se utiliza para almacenar valores enteros de 0 a 32767 (`2 Bytes`). Al igual que SmallIntegerField, este campo es útil para valores que no son tan altos y debería ser semánticamente positivo. Por ejemplo, puede almacenar la edad que no puede ser negativa.

```
from django.db import models

class Staff(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    age = models.PositiveSmallIntegerField(null=True)
```

Además de PositiveSmallIntegerField es útil para las opciones, esta es la forma Djangoic de implementar Enum:

```
from django.db import models
from django.utils.translation import gettext as _

APPLICATION_NEW = 1
APPLICATION_RECEIVED = 2
APPLICATION_APPROVED = 3
APPLICATION_REJECTED = 4

APLICATION_CHOICES = (
    (APPLICATION_NEW, _('New')),
    (APPLICATION_RECEIVED, _('Received')),
    (APPLICATION_APPROVED, _('Approved')),
    (APPLICATION_REJECTED, _('Rejected')),
)

class JobApplication(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    status = models.PositiveSmallIntegerField(
        choices=APLICATION_CHOICES,
        default=APPLICATION_NEW
    )
    ...
```

> La definición de las opciones como variables de clase o variables de módulo según la situación es una buena manera de usarlas. Si se pasan opciones al campo sin nombres amistosos, se creará confusión.

Campo decimal

Un número decimal de precisión fija, representado en Python por una instancia decimal. A diferencia de IntegerField y sus derivados, este campo tiene 2 argumentos requeridos:

1. *DecimalField.max_digits* : el número máximo de dígitos permitido en el número. Tenga en cuenta que este número debe ser mayor o igual que decimal_places.
2. *DecimalField.decimal_places* : El número de lugares decimales para almacenar con el número.

Si desea almacenar números de hasta 99 con 3 lugares decimales necesita usar `max_digits=5` y `decimal_places=3` :

```
class Place(models.Model):
    name = models.CharField(max_length=255)
    atmospheric_pressure = models.DecimalField(max_digits=5, decimal_places=3)
```

Campo binario

Este es un campo especializado, utilizado para almacenar datos binarios. Solo acepta **bytes** . Los datos son base64 serializados en el almacenamiento.

Como esto es almacenar datos binarios, este campo no se puede usar en un filtro.

```
from django.db import models

class MyModel(models.Model):
    my_binary_data = models.BinaryField()
```

Campo de golf

CharField se utiliza para almacenar longitudes de texto definidas. En el siguiente ejemplo, se pueden almacenar hasta 128 caracteres de texto en el campo. Si ingresa una cadena más larga que esto, se generará un error de validación.

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=128, blank=True)
```

DateTimeField

DateTimeField se utiliza para almacenar valores de fecha y hora.

```
class MyModel(models.Model):
    start_time = models.DateFimeField(null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)
```

Un `DateTimeField` tiene dos parámetros opcionales:

- `auto_now_add` establece el valor del campo en la fecha y hora actual cuando se crea el objeto.
- `auto_now` establece el valor del campo en la fecha y hora actual cada vez que se guarda el campo.

Estas opciones y el parámetro `default` se excluyen mutuamente.

Clave externa

El campo ForeignKey se utiliza para crear una relación de `many-to-one` entre los modelos. No como la mayoría de los otros campos requiere argumentos posicionales. El siguiente ejemplo demuestra la relación de automóvil y propietario:

```
from django.db import models

class Person(models.Model):
    GENDER_FEMALE = 'F'
    GENDER_MALE = 'M'

    GENDER_CHOICES = (
        (GENDER_FEMALE, 'Female'),
        (GENDER_MALE, 'Male'),
    )

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
    age = models.SmallIntegerField()


class Car(model.Model)
    owner = models.ForeignKey('Person')
    plate = models.CharField(max_length=15)
    brand = models.CharField(max_length=50)
    model = models.CharField(max_length=50)
    color = models.CharField(max_length=50)
```

El primer argumento del campo es la clase con la que se relaciona el modelo. El segundo argumento posicional es el argumento `on_delete` . En las versiones actuales, este argumento no es necesario, pero será necesario en Django 2.0. La funcionalidad predeterminada del argumento se muestra a continuación:

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.CASCADE)
    ...
```

> Esto provocará que los objetos de automóvil se eliminen del modelo cuando su propietario se elimine del modelo de persona. Esta es la funcionalidad por defecto.

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.PROTECT)
```

```
...
```

Esto evitará que los objetos Person se eliminen si están relacionados con al menos un objeto Car. Todos los objetos de Car que hacen referencia a un objeto de Persona deben eliminarse primero. Y luego el objeto de persona puede ser eliminado.

Lea Modelo de referencia de campo en línea: https://riptutorial.com/es/django/topic/3686/modelo-de-referencia-de-campo

Capítulo 39: Modelos

Introducción

En el caso básico, un modelo es una clase de Python que se asigna a una única tabla de base de datos. Los atributos de la clase se asignan a las columnas de la tabla y una instancia de la clase representa una fila en la tabla de la base de datos. Los modelos heredan de `django.db.models.Model` que proporciona una API enriquecida para agregar y filtrar resultados de la base de datos.

Crea tu primer modelo

Examples

Creando tu primer modelo

Los modelos normalmente se definen en el archivo `models.py` en el subdirectorio de su aplicación. El `Model` la clase de `django.db.models` módulo es una buena clase de partida para extender sus modelos de. Por ejemplo:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey('Author', on_delete=models.CASCADE,
related_name='authored_books')
    publish_date = models.DateField(null=True, blank=True)

    def __str__(self): # __unicode__ in python 2.*
        return self.title
```

Cada atributo en un modelo representa una columna en la base de datos.

- `title` es un texto con una longitud máxima de 100 caracteres.
- `author` es una `ForeignKey` que representa una relación con otro modelo / tabla, en este caso `Author` (solo se utiliza con fines de ejemplo). `on_delete` le dice a la base de datos qué hacer con el objeto en caso de que se elimine el objeto relacionado (un `Author`). (Cabe señalar que dado que django 1.9 `on_delete` puede usarse como el segundo argumento posicional. En django 2 es un argumento **obligatorio** y es recomendable tratarlo como tal. En versiones anteriores, se usará de forma predeterminada en `CASCADE` .)
- `publish_date` almacena una fecha. Tanto el `null` como el `blank` se establecen en `True` para indicar que no es un campo obligatorio (es decir, puede agregarlo en una fecha posterior o dejarlo vacío).

Junto con los atributos, definimos un método `__str__` devuelve el título del libro que se usará como su representación de `string` cuando sea necesario, en lugar del valor predeterminado.

Aplicando los cambios a la base de datos (Migraciones).

Después de crear un nuevo modelo o modificar modelos existentes, deberá generar migraciones para sus cambios y luego aplicar las migraciones a la base de datos especificada. Esto se puede hacer usando el sistema de migraciones incorporado de Django. Usando la utilidad `manage.py` cuando `manage.py` en el directorio raíz del proyecto:

```
python manage.py makemigrations <appname>
```

El comando anterior creará los scripts de migración que son necesarios en el subdirectorio de `migrations` de su aplicación. Si omite el parámetro `<appname>` , se procesarán todas las aplicaciones definidas en el argumento `INSTALLED_APPS` de `settings.py` . Si lo considera necesario, puede editar las migraciones.

Puede verificar qué migraciones son necesarias sin crear realmente la migración, use la opción --dry-run, por ejemplo:

```
python manage.py makemigrations --dry-run
```

Para aplicar las migraciones:

```
python manage.py migrate <appname>
```

El comando anterior ejecutará los scripts de migración generados en el primer paso y actualizará físicamente la base de datos.

Si se cambia el modelo de la base de datos existente, se necesita el siguiente comando para realizar los cambios necesarios.

```
python manage.py migrate --run-syncdb
```

Django creará la tabla con el nombre `<appname>_<classname>` de forma predeterminada. En algún momento no quieres usarlo. Si desea cambiar el nombre predeterminado, puede anunciar el nombre de la tabla configurando `db_table` en la clase `Meta` :

```
from django.db import models

class YourModel(models.Model):
    parms = models.CharField()
    class Meta:
        db_table = "custom_table_name"
```

Si desea ver qué código SQL ejecutará una determinada migración, simplemente ejecute este comando:

```
python manage.py sqlmigrate <app_label> <migration_number>
```

Django> 1.10

La nueva opción `makemigrations --check` hace que el comando salga con un estado distinto de cero cuando se detectan cambios en el modelo sin migraciones.

Ver Migraciones para más detalles sobre migraciones.

Creando un modelo con relaciones.

Relación de muchos a uno

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

#Book has a foreignkey (many to one) relationship with author
class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    publish_date = models.DateField()
```

La opción más genérica. Se puede utilizar en cualquier lugar que desee representar una relación

Relación de muchos a muchos

```
class Topping(models.Model):
    name = models.CharField(max_length=50)

# One pizza can have many toppings and same topping can be on many pizzas
class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)
```

Internamente esto se representa a través de otra tabla. Y `ManyToManyField` debe colocar en modelos que se `ManyToManyField` en un formulario. Por ejemplo: la `Appointment` tendrá un `ManyToManyField` llamado `Customer` , `Pizza` tiene `Toppings` y así sucesivamente.

Relación de muchos a muchos usando clases directas

```
class Service(models.Model):
     name = models.CharField(max_length=35)

class Client(models.Model):
    name = models.CharField(max_length=35)
    age = models.IntegerField()
    services = models.ManyToManyField(Service, through='Subscription')

class Subscription(models.Model):
     client = models.ForeignKey(Client)
     service = models.ForeignKey(Service)
     subscription_type = models.CharField(max_length=1, choices=SUBSCRIPTION_TYPES)
     created_at = models.DateTimeField(default=timezone.now)
```

De esta manera, podemos mantener más metadatos sobre una relación entre dos entidades. Como puede verse, un cliente puede suscribirse a varios servicios a través de varios tipos de suscripción. La única diferencia en este caso es que para agregar nuevas instancias a la relación

M2M, no se puede usar el método abreviado `pizza.toppings.add(topping)` , sino que se debe crear un nuevo objeto de la clase *through* , `Subscription.objects.create(client=client, service=service, subscription_type='p')`

> En otros idiomas, las `through tables` también se conocen como `JoinColumn` , `Intersection table` o `mapping table`

Relación uno a uno

```
class Employee(models.Model):
   name = models.CharField(max_length=50)
   age = models.IntegerField()
   spouse = models.OneToOneField(Spouse)

class Spouse(models.Model):
   name = models.CharField(max_length=50)
```

Utilice estos campos cuando solo tendrá una relación de composición entre los dos modelos.

Consultas básicas de Django DB

Django ORM es una poderosa abstracción que le permite almacenar y recuperar datos de la base de datos sin tener que escribir consultas SQL por su cuenta.

Asumamos los siguientes modelos:

```
class Author(models.Model):
   name = models.CharField(max_length=50)

class Book(models.Model):
   name = models.CharField(max_length=50)
   author = models.ForeignKey(Author)
```

Suponiendo que haya agregado el código anterior a una aplicación de django y ejecute el comando `migrate` (para que se cree su base de datos). Inicia el shell Django

```
python manage.py shell
```

Esto inicia el shell estándar de Python, pero con las bibliotecas de Django relevantes importadas, para que pueda centrarse directamente en las partes importantes.

Comience importando los modelos que acabamos de definir (supongo que esto se hace en un archivo `models.py`)

```
from .models import Book, Author
```

Ejecute su primera consulta de selección:

```
>>> Author.objects.all()
[]
>>> Book.objects.all()
```

```
[]
```

Permite crear un autor y objeto de libro:

```
>>> hawking = Author(name="Stephen hawking")
>>> hawking.save()
>>> history_of_time = Book(name="history of time", author=hawking)
>>> history_of_time.save()
```

o use la función crear para crear objetos modelo y guardar en un código de línea

```
>>> wings_of_fire = Book.objects.create(name="Wings of Fire", author="APJ Abdul Kalam")
```

Ahora vamos a ejecutar la consulta

```
>>> Book.objects.all()
[<Book: Book object>]
>>> book = Book.objects.first() #getting the first book object
>>> book.name
u'history of time'
```

Agreguemos una cláusula where a nuestra consulta de selección

```
>>> Book.objects.filter(name='nothing')
[]
>>> Author.objects.filter(name__startswith='Ste')
[<Author: Author object>]
```

Para obtener los detalles sobre el autor de un libro determinado

```
>>> book = Book.objects.first() #getting the first book object
>>> book.author.name # lookup on related model
u'Stephen hawking'
```

Para obtener todos los libros publicados por Stephen Hawking (libro de búsqueda por su autor)

```
>>> hawking.book_set.all()
[<Book: Book object>]
```

`_set` es la notación que se usa para "búsquedas inversas", es decir, mientras el campo de búsqueda está en el modelo de Libro, podemos usar `book_set` en un objeto de autor para obtener todos sus libros.

Una tabla básica no gestionada.

En algún momento del uso de Django, es posible que desee interactuar con tablas que ya se han creado o con vistas de base de datos. En estos casos, no querrá que Django administre las tablas a través de sus migraciones. Para configurar esto, necesita agregar solo una variable a la clase `Meta` su modelo: `managed = False` .

Este es un ejemplo de cómo puede crear un modelo no administrado para interactuar con una vista de base de datos:

```
class Dummy(models.Model):
    something = models.IntegerField()

    class Meta:
       managed = False
```

Esto se puede asignar a una vista definida en SQL de la siguiente manera.

```
CREATE VIEW myapp_dummy AS
SELECT id, something FROM complicated_table
WHERE some_complicated_condition = True
```

Una vez que haya creado este modelo, puede usarlo como lo haría con cualquier otro modelo:

```
>>> Dummy.objects.all()
[<Dummy: Dummy object>, <Dummy: Dummy object>, <Dummy: Dummy object>]
>>> Dummy.objects.filter(something=42)
[<Dummy: Dummy object>]
```

Modelos avanzados

Un modelo puede proporcionar mucha más información que solo los datos sobre un objeto. Veamos un ejemplo y dividámoslo en lo que es útil para:

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    slug = models.SlugField()
    title = models.CharField(max_length=128)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('library:book', kwargs={'pk':self.pk})

    def __str__(self):
        return self.title

    class Meta:
        ordering = ['publish_date', 'title']
```

Llave primaria automatica

Puede notar el uso de `self.pk` en el método `get_absolute_url` . El campo `pk` es un alias de la clave principal de un modelo. Además, django agregará automáticamente una clave principal si falta. Eso es una cosa menos de la que preocuparse: le permite establecer una clave externa para cualquier modelo y obtenerla fácilmente.

Url absoluta

La primera función que se define es `get_absolute_url` . De esta manera, si tiene un libro, puede obtener un enlace a él sin tener que manipular la etiqueta url, la resolución, el atributo y similares. Simplemente llame a `book.get_absolute_url` y obtendrá el enlace correcto. Como beneficio adicional, su objeto en el administrador de django obtendrá un botón "ver en el sitio".

Representación de cuerdas

Tenga un método `__str__` que le permita usar el objeto cuando necesite mostrarlo. Por ejemplo, con el método anterior, agregar un enlace al libro en una plantilla es tan simple como `<a href="{{ book.get_absolute_url }}">{{ book }}</a>` . Directo al grano. Este método también controla lo que se muestra en el menú desplegable de administración, por ejemplo, para clave externa.

El decorador de clase te permite definir el método una vez para `__str__` y `__unicode__` en python 2 sin causar problemas en python 3. Si esperas que tu aplicación se ejecute en ambas versiones, esa es la manera de hacerlo.

Campo de babosa

El campo slug es similar a un campo char pero acepta menos símbolos. Por defecto, solo letras, números, guiones bajos o guiones. Es útil si desea identificar un objeto con una buena representación, por ejemplo, en url.

La clase meta

La clase `Meta` nos permite definir mucha más información sobre toda la colección de artículos. Aquí, solo se establece el orden predeterminado. Es útil con el objeto ListView, por ejemplo. Se necesita una lista corta ideal de campos para usar en la clasificación. Aquí, el libro se ordenará primero por fecha de publicación y luego por título si la fecha es la misma.

Otros atributos frecuentes son `verbose_name` y `verbose_name_plural` . De forma predeterminada, se generan a partir del nombre del modelo y deben estar bien. Pero la forma plural es ingenua, simplemente anexando una 's' al singular por lo que es posible que desee establecerlo explícitamente en algún caso.

Valores calculados

Una vez que se ha recuperado un objeto modelo, se convierte en una instancia completamente realizada de la clase. Como tal, se puede acceder a cualquier método adicional en formularios y serializadores (como Django Rest Framework).

El uso de propiedades de python es una forma elegante de representar valores adicionales que no se almacenan en la base de datos debido a diversas circunstancias.

```
def expire():
    return timezone.now() + timezone.timedelta(days=7)

class Coupon(models.Model):
    expiration_date = models.DateField(default=expire)

    @property
    def is_expired(self):
        return timezone.now() > self.expiration_date
```

Si bien la mayoría de los casos puede complementar los datos con anotaciones en sus consultas, los valores computados como propiedades de modelo son ideales para cálculos que no pueden evaluarse simplemente dentro del alcance de una consulta.

Además, las propiedades, ya que están declaradas en la clase python y no como parte del esquema, no están disponibles para realizar consultas.

Añadiendo una representación de cadena de un modelo.

Para crear una presentación legible por humanos de un objeto modelo, necesita implementar el método `Model.__str__()` (o `Model.__unicode__()` en python2). Se llamará a este método siempre que llame a `str()` en una instancia de su modelo (incluido, por ejemplo, cuando el modelo se utiliza en una plantilla). Aquí hay un ejemplo:

1. Crear un modelo de libro.

```
# your_app/models.py

from django.db import models

class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)
```

2. Cree una instancia del modelo y guárdela en la base de datos:

```
>>> himu_book = Book(name='Himu Mama', author='Humayun Ahmed')
>>> himu_book.save()
```

3. Ejecute `print()` en la instancia:

```
>>> print(himu_book)
<Book: Book object>
```

<Libro: Objeto de libro> , el resultado predeterminado, no nos ayuda. Para solucionar esto, vamos a agregar un método `__str__` .

```
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    name = models.CharField(max_length=50)
```

```
    author = models.CharField(max_length=50)

    def __str__(self):
        return '{} by {}'.format(self.name, self.author)
```

Tenga en cuenta que el decorador compatible con `python_2_unicode_compatible` es necesario solo si desea que su código sea compatible con python 2. Este decorador copia el método `__str__` para crear un método `__unicode__` . Importarlo desde `django.utils.encoding` .

Ahora si llamamos a la función de impresión la instancia de libro de nuevo:

```
>>> print(himu_book)
Himu Mama by Humayun Ahmed
```

¡Mucho mejor!

La representación de la cadena también se usa cuando el modelo se usa en los `ModelForm` para `ForeignKeyField` y `ManyToManyField` .

Modelo mixins

En los mismos casos, diferentes modelos podrían tener los mismos campos y procedimientos en el ciclo de vida del producto. Para manejar estas similitudes sin tener herencia de repetición de código se podría utilizar. En lugar de heredar una clase completa, el patrón de diseño **mixin** nos ofrece heredar (*o algunos dicen que incluyen*) algunos métodos y atributos. Veamos un ejemplo:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    sender_name = models.CharField(max_length=128)
    sender_address = models.CharField(max_length=255)
    receiver_name = models.CharField(max_length=128)
    receiver_address = models.CharField(max_length=255)
    post_datetime = models.DateTimeField(auto_now_add=True)
    delivery_datetime = models.DateTimeField(null=True)
    notes = models.TextField(max_length=500)

class Envelope(PostableMixin):
    ENVELOPE_COMMERCIAL = 1
    ENVELOPE_BOOKLET = 2
    ENVELOPE_CATALOG = 3

    ENVELOPE_TYPES = (
        (ENVELOPE_COMMERCIAL, 'Commercial'),
        (ENVELOPE_BOOKLET, 'Booklet'),
        (ENVELOPE_CATALOG, 'Catalog'),
    )

    envelope_type = models.PositiveSmallIntegerField(choices=ENVELOPE_TYPES)

class Package(PostableMixin):
    weight = models.DecimalField(max_digits=6, decimal_places=2)
    width = models.DecimalField(max_digits=5, decimal_places=2)
    height = models.DecimalField(max_digits=5, decimal_places=2)
```

```
    depth = models.DecimalField(max_digits=5, decimal_places=2)
```

Para convertir un modelo en una clase abstracta, deberá mencionar `abstract=True` en su clase `Meta` interna. Django no crea ninguna tabla para modelos abstractos en la base de datos. Sin embargo, para los modelos `Envelope` y `Package` , las tablas correspondientes se crearían en la base de datos.

Además, los campos de algunos métodos de modelo serán necesarios en más de un modelo. Por lo tanto, estos métodos podrían agregarse a mixins para evitar la repetición del código. Por ejemplo, si creamos un método para establecer la fecha de entrega en `PostableMixin` , será accesible desde sus dos hijos:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    ...
    ...

    def set_delivery_datetime(self, dt=None):
        if dt is None:
            from django.utils.timezone import now
            dt = now()

        self.delivery_datetime = dt
        self.save()
```

Este método se puede utilizar como sigue en los niños:

```
>> envelope = Envelope.objects.get(pk=1)
>> envelope.set_delivery_datetime()

>> pack = Package.objects.get(pk=1)
>> pack.set_delivery_datetime()
```

UUID clave primaria

Un modelo por defecto utilizará una clave primaria de incremento automático (entero). Esto le dará una secuencia de teclas 1, 2, 3.

Se pueden establecer diferentes tipos de clave primaria en un modelo con pequeñas modificaciones al modelo.

Un UUID es un identificador único universal, este es un identificador aleatorio de 32 caracteres que se puede usar como una identificación. Esta es una buena opción para usar cuando no desea que los ID secuenciales se asignen a los registros en su base de datos. Cuando se usa en PostgreSQL, esto se almacena en un tipo de datos uuid, de lo contrario en un char (32).

```
import uuid
from django.db import models

class ModelUsingUUID(models.Model):
```

```
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
```

La clave generada estará en el formato `7778c552-73fc-4bc4-8bf9-5a2f6f7b7f47`

Herencia

La herencia entre modelos se puede hacer de dos maneras:

- una clase abstracta común (ver el ejemplo "Modelo mixins")
- un modelo común con múltiples tablas

La herencia de tablas múltiples creará una tabla para los campos comunes y una por ejemplo de modelo secundario:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

creará 2 tablas, una para `Place` y otra para `Restaurant` con un campo `OneToOne` oculto a `Place` para los campos comunes.

tenga en cuenta que esto requerirá una consulta adicional a las tablas de lugares cada vez que obtenga un objeto de restaurante.

Lea Modelos en línea: https://riptutorial.com/es/django/topic/888/modelos

Capítulo 40: Plantilla

Examples

Variables

Se puede acceder a las variables que ha proporcionado en el contexto de su vista usando la notación de doble refuerzo:

En tus `views.py` :

```
class UserView(TemplateView):
  """ Supply the request user object to the template """

  template_name = "user.html"

  def get_context_data(self, **kwargs):
    context = super(UserView, self).get_context_data(**kwargs)
    context.update(user=self.request.user)
    return context
```

En `user.html` :

```
<h1>{{ user.username }}</h1>

<div class="email">{{ user.email }}</div>
```

La notación de puntos tendrá acceso:

- propiedades del objeto, por ejemplo, `user.username` será `{{ user.username }}`
- búsquedas de diccionario, por ejemplo, `request.GET["search"]` será `{{ request.GET.search }}`
- los métodos sin argumentos, por ejemplo, `users.count()` serán `{{ user.count }}`

Las variables de la plantilla no pueden acceder a los métodos que toman argumentos.

Las variables también pueden ser probadas y repartidas en:

```
{% if user.is_authenticated %}
  {% for item in menu %}
    <li><a href="{{ item.url }}">{{ item.name }}</a></li>
  {% endfor %}
{% else %}
  <li><a href="{% url 'login' %}">Login</a>
{% endif %}
```

Se accede a las URL utilizando el formato `{% url 'name' %}` , donde los nombres corresponden a los nombres en `urls.py`

`{% url 'login' %}` - Probablemente se procesará como `/accounts/login/`

`{% url 'user_profile' user.id %}` - Los argumentos para las URL se proporcionan en orden

`{% url next %}` - Las URL pueden ser variables

Plantillas en vistas basadas en clase

Puede pasar datos a una plantilla en una variable personalizada.

En tus `views.py` :

```
from django.views.generic import TemplateView
from MyProject.myapp.models import Item

class ItemView(TemplateView):
    template_name = "item.html"

    def items(self):
        """ Get all Items """
        return Item.objects.all()

    def certain_items(self):
        """ Get certain Items """
        return Item.objects.filter(model_field="certain")

    def categories(self):
        """ Get categories related to this Item """
        return Item.objects.get(slug=self.kwargs['slug']).categories.all()
```

Una lista simple en tu `item.html` :

```
{% for item in view.items %}
<ul>
    <li>{{ item }}</li>
</ul>
{% endfor %}
```

También puede recuperar propiedades adicionales de los datos.

Asumiendo que su `Item` modelo tiene un campo de `name` :

```
{% for item in view.certain_items %}
<ul>
    <li>{{ item.name }}</li>
</ul>
{% endfor %}
```

Plantillas en vistas basadas en funciones

Puede utilizar una plantilla en una vista basada en funciones de la siguiente manera:

```
from django.shortcuts import render

def view(request):
    return render(request, "template.html")
```

Si desea utilizar variables de plantilla, puede hacerlo de la siguiente manera:

```
from django.shortcuts import render

def view(request):
    context = {"var1": True, "var2": "foo"}
    return render(request, "template.html", context=context)
```

Luego, en `template.html` , puede referirse a sus variables así:

```
<html>
{% if var1 %}
    <h1>{{ var2 }}</h1>
{% endif %}
</html>
```

Filtros de plantillas

El sistema de plantillas de Django tiene *etiquetas* y *filtros* incorporados, que son funciones dentro de la plantilla para representar contenido de una manera específica. Se pueden especificar varios filtros con canalizaciones y los filtros pueden tener argumentos, al igual que en la sintaxis variable.

```
{{ "MAINROAD 3222"|lower }}    # mainroad 3222
{{ 10|add:15}}                 # 25
{{ "super"|add:"glue" }}       # superglue
{{ "A7"|add:"00" }}            # A700
{{ myDate | date:"D d M Y"}}   # Wed 20 Jul 2016
```

Puede encontrar una lista de los **filtros integrados** disponibles en https://docs.djangoproject.com/en/dev/ref/templates/builtins/#ref-templates-builtins-filters .

Creando filtros personalizados

Para agregar sus propios filtros de plantilla, cree una carpeta llamada `templatetags` dentro de la carpeta de su aplicación. Luego agregue un `__init__.py` , y el archivo de su archivo que contendrá los filtros:

```
#/myapp/templatetags/filters.py
from django import template

register = template.Library()

@register.filter(name='tostring')
def to_string(value):
    return str(value)
```

Para usar realmente el filtro necesitas cargarlo en tu plantilla:

```
#templates/mytemplate.html
{% load filters %}
{% if customer_id|tostring = customer %} Welcome back {% endif%}
```

Trucos

A pesar de que los filtros parecen simples al principio, permite hacer algunas cosas ingeniosas:

```
{% for x in ""|ljust:"20" %}Hello World!{% endfor %}    # Hello World!Hello World!Hel...
{{ user.name.split|join:"_" }} ## replaces whitespace with '_'
```

Ver también etiquetas de plantillas para más información.

Evitar que los métodos sensibles sean llamados en plantillas

Cuando un objeto se expone al contexto de la plantilla, sus métodos sin argumentos están disponibles. Esto es útil cuando estas funciones son "captadores". Pero puede ser peligroso si estos métodos alteran algunos datos o tienen algunos efectos secundarios. Aunque es probable que confíe en el autor de la plantilla, es posible que no esté al tanto de los efectos secundarios de una función o que piense que debe llamar al atributo incorrecto por error.

Dado el siguiente modelo:

```
class Foobar(models.Model):
    points_credit = models.IntegerField()

    def credit_points(self, nb_points=1):
        """Credit points and return the new points credit value."""
        self.points_credit = F('points_credit') + nb_points
        self.save(update_fields=['points_credit'])
        return self.points_credit
```

Si escribes esto, por error, en una plantilla:

```
 You have {{ foobar.credit_points }} points!
```

Esto incrementará el número de puntos cada vez que se llame a la plantilla. Y puede que ni siquiera lo note.

Para evitar esto, debe establecer el atributo `alters_data` en `True` a los métodos que tienen efectos secundarios. Esto hará que sea imposible llamarlos desde una plantilla.

```
def credit_points(self, nb_points=1):
    """Credit points and return the new points credit value."""
    self.points_credit = F('points_credit') + nb_points
    self.save(update_fields=['points_credit'])
    return self.points_credit
credit_points.alters_data = True
```

El uso de {% extiende%}, {% incluye%} y {% bloques%}

resumen

- **{% extiende%}** : esto declara que la plantilla dada como argumento es la principal de la plantilla actual. Uso: `{% extends 'parent_template.html' %}` .
- **{% block%} {% endblock%}** : esto se usa para definir secciones en sus plantillas, de modo que si otra plantilla extiende esta, podrá reemplazar cualquier código html que se haya escrito dentro de ella. Los bloques se identifican por su nombre. Uso: `{% block content %} <html_code> {% endblock %}` .
- **{% include%}** : esto insertará una plantilla dentro de la actual. Tenga en cuenta que la plantilla incluida recibirá el contexto de la solicitud y que también puede asignarle variables personalizadas. Uso básico: `{% include 'template_name.html' %}` , uso con variables: `{% include 'template_name.html' with variable='value' variable2=8 %}`

Guía

Supongamos que está creando su código del lado frontal con diseños comunes para todos los códigos y no desea repetir el código para cada plantilla. Django te da en las etiquetas construidas para hacerlo.
Supongamos que tenemos un sitio web de blog con 3 plantillas que comparten el mismo diseño:

```
project_directory
    ..
    templates
      front-page.html
      blogs.html
      blog-detail.html
```

1) Definir el archivo `base.html` ,

```
<html>
  <head>
  </head>

  <body>
        {% block content %}
        {% endblock %}
   </body>
</html>
```

2) `blog.html` en `blog.html` como,

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

{% block content %}
    # write your blog related code here
{% endblock %}

# None of the code written here will be added to the template
```

Aquí `blog.html` el diseño base para que su diseño HTML ahora esté disponible en el `blog.html` blog.html. El concepto de `{ % block %}` es la herencia de la plantilla que le permite crear una

plantilla básica "esqueleto" que contiene todos los elementos comunes de su sitio y define los bloques que las plantillas secundarias pueden anular.

3) Ahora suponga que todas sus 3 plantillas también tienen la misma división de HTML que define algunas publicaciones populares. En lugar de escribirse 3 veces, cree una nueva plantilla `posts.html` .

blog.html

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

{% block content %}
    # write your blog related code here
    {% include 'posts.html' %} # includes posts.html in blog.html file without passing any
data
    <!-- or -->
    {% include 'posts.html' with posts=postdata %} # includes posts.html in blog.html file
with passing posts data which is context of view function returns.
{% endblock %}
```

Lea Plantilla en línea: https://riptutorial.com/es/django/topic/588/plantilla

Capítulo 41: Procesadores de contexto

Observaciones

Use procesadores de contexto para agregar variables que sean accesibles en cualquier parte de sus plantillas.

Especifique una función, o funciones que devuelvan `dict` de las variables que desea, luego agregue esas funciones a `TEMPLATE_CONTEXT_PROCESSORS` .

Examples

Utilice un procesador de contexto para acceder a settings.DEBUG en plantillas

en `myapp/context_processors.py` :

```
from django.conf import settings

def debug(request):
  return {'DEBUG': settings.DEBUG}
```

en `settings.py` :

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.debug',
            ],
        },
    },
]
```

o, para las versiones <1.9:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'myapp.context_processors.debug',
)
```

Luego en mis plantillas, simplemente:

```
 {% if DEBUG %} .header { background:#f00; } {% endif %}
 {{ DEBUG }}
```

Uso de un procesador de contexto para acceder a las entradas de blog más recientes en todas las plantillas

Suponiendo que tiene un modelo llamado `Post` definida en su archivo `models.py` que contiene publicaciones de blog y tiene un campo de fecha de `date_published` .

Paso 1: Escribe el procesador de contexto

Cree (o agregue a) un archivo en el directorio de su aplicación llamado `context_processors.py` :

```
from myapp.models import Post

def recent_blog_posts(request):
    return {'recent_posts':Post.objects.order_by('-date_published')[0:3],}  # Can change
numbers for more/fewer posts
```

Paso 2: Agrega el procesador de contexto a tu archivo de configuración

Asegúrese de agregar su nuevo procesador de contexto a su archivo `settings.py` en la variable `TEMPLATES` :

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.recent_blog_posts',
            ],
        },
    },
]
```

(En las versiones de Django anteriores a 1.9, esto se configuró directamente en `settings.py` utilizando una variable `TEMPLATE_CONTEXT_PROCESSORS`).

Paso 3: Usa el procesador de contexto en tus plantillas

¡Ya no es necesario pasar entradas de blog recientes a través de vistas individuales! Solo usa `recent_blog_posts` en cualquier plantilla.

Por ejemplo, en `home.html` puede crear una barra lateral con enlaces a publicaciones recientes:

```
<div class="blog_post_sidebar">
    {% for post in recent_blog_posts %}
        <div class="post">
            <a href="{{post.get_absolute_url}}">{{post.title}}</a>
        </div>
    {% endfor %}
</div>
```

O en `blog.html` puede crear una visualización más detallada de cada publicación:

```
<div class="content">
    {% for post in recent_blog_posts %}
        <div class="post_detail">
            <h2>{{post.title}}</h2>
            <p>Published on {{post.date_published}}</p>
            <p class="author">Written by: {{post.author}}</p>
            <p><a href="{{post.get_absolute_url}}">Permalink</a></p>
            <p class="post_body">{{post.body}}</p>
        </div>
    {% endfor %}
</div>
```

Extendiendo tus plantillas

Procesador de contexto para determinar la plantilla según la pertenencia al grupo (o cualquier consulta / lógica). Esto permite a nuestros usuarios públicos / regulares obtener una plantilla y nuestro grupo especial obtener una plantilla diferente.

myapp / context_processors.py

```
def template_selection(request):
    site_template = 'template_public.html'
    if request.user.is_authenticated():
        if request.user.groups.filter(name="some_group_name").exists():
            site_template = 'template_new.html'

    return {
        'site_template': site_template,
    }
```

Agregue el procesador de contexto a su configuración.

En sus plantillas, use la variable definida en el procesador de contexto.

```
{% extends site_template %}
```

Lea Procesadores de contexto en línea: https://riptutorial.com/es/django/topic/491/procesadores-de-contexto

Capítulo 42: Puntos de vista

Introducción

Una función de vista, o vista para abreviar, es simplemente una función de Python que toma una solicitud web y devuelve una respuesta web. -Django Documentación-

Examples

[Introducción] Vista simple (Hello World Equivalent)

Vamos a crear una vista muy simple para responder a una plantilla "Hello World" en formato html.

1. Para hacerlo, vaya a `my_project/my_app/views.py` (Aquí estamos alojando nuestras funciones de vista) y agregue la siguiente vista:

```
from django.http import HttpResponse

def hello_world(request):
    html = "<html><title>Hello World!</title><body>Hello World!</body></html>"
    return HttpResponse(html)
```

2. Para llamar a esta vista, necesitamos configurar un patrón de url en `my_project/my_app/urls.py` :

```
from django.conf.urls import url

from . import views

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

3. Inicia el servidor: `python manage.py runserver`

   Ahora, si `http://localhost:8000/hello_world/` , nuestra plantilla (la cadena html) se representará en nuestro navegador.

Lea Puntos de vista en línea: https://riptutorial.com/es/django/topic/7490/puntos-de-vista

Capítulo 43: Querysets

Introducción

Un `Queryset` es fundamentalmente una lista de objetos derivados de un `Model` , por una compilación de consultas de base de datos.

Examples

Consultas simples en un modelo independiente.

Aquí hay un modelo simple que usaremos para ejecutar algunas consultas de prueba:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Consigue un objeto modelo único donde el id / pk es 4:
(Si no hay elementos con el ID de 4 o hay más de uno, esto generará una excepción).

```
MyModel.objects.get(pk=4)
```

Todos los objetos modelo:

```
MyModel.objects.all()
```

Objetos de modelo que tienen el `flag` establecido en `True` :

```
MyModel.objects.filter(flag=True)
```

Objetos de modelo con un `model_num` mayor que 25:

```
MyModel.objects.filter(model_num__gt=25)
```

Objetos de modelo con el `name` de "Artículo barato" y `flag` establecida en `False` :

```
MyModel.objects.filter(name="Cheap Item", flag=False)
```

Modelos de `name` búsqueda simple para una cadena específica (distingue entre mayúsculas y minúsculas)

```
MyModel.objects.filter(name__contains="ch")
```

Modelos de búsqueda simple de `name` para una cadena específica (no distingue mayúsculas y

minúsculas):

```
MyModel.objects.filter(name__icontains="ch")
```

Consultas avanzadas con objetos Q

Dado el modelo:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Podemos usar objetos `Q` para crear condiciones `AND` , `OR` en su consulta de búsqueda. Por ejemplo, supongamos que queremos que todos los objetos que tienen el `flag=True` **O** `model_num>15` .

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15))
```

Lo anterior se traduce a `WHERE flag=True OR model_num > 15` manera similar para un **AND** que haría.

```
MyModel.objects.filter(Q(flag=True) & Q(model_num__gt=15))
```

`Q` objetos `Q` también nos permiten hacer consultas **NO** con el uso de `~` . Digamos que queríamos obtener todos los objetos que tienen `flag=False` **AND** `model_num!=15` , haríamos:

```
MyModel.objects.filter(Q(flag=True) & ~Q(model_num=15))
```

Si se usan objetos Q y parámetros "normales" en el `filter()` , entonces los objetos `Q` deben aparecer *primero* . La siguiente consulta busca modelos con (`flag` establecida en `True` o un número de modelo mayor que `15`) y un nombre que comience con "H".

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15), name__startswith="H")
```

Nota: los objetos `Q` se pueden usar con cualquier función de búsqueda que tome argumentos de palabras clave como `filter` , `exclude` , `get` . Asegúrese de que cuando use con `get` solo devolverá un objeto o se generará la excepción `MultipleObjectsReturned` .

Reducir el número de consultas en ManyToManyField (problema n + 1)

Problema

```
# models.py:
class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.all()

    # Query the database on each iteration (len(author) times)
    # if there is 100 librairies, there will have 100 queries plus the initial query
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 101 queries
```

Solución

Use `prefetch_related` en `ManyToManyField` si sabe que necesitará acceder más tarde a un campo que es un campo `ManyToManyField` .

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()

    # Does not query the database again, since `books` is pre-populated
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 2 queries - 1 for libraries, 1 for books
```

`prefetch_related` también se puede utilizar en los campos de búsqueda:

```
# models.py:
class User(models.Model):
    name = models.CharField(max_length=100)

class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)

class Book(models.Model):
    title = models.CharField(max_length=100)
    readers = models.ManyToManyField(User)
```

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books', 'books__readers').all()
```

```
    # Does not query the database again, since `books` and `readers` is pre-populated
    for library in libraries:
        for book in library.books.all():
            for user in book.readers.all():
                user.name
                # ...

    # total : 3 queries - 1 for libraries, 1 for books, 1 for readers
```

Sin embargo, una vez que se ha ejecutado el queryset, los datos obtenidos no pueden alterarse sin volver a golpear la base de datos. Lo siguiente ejecutaría consultas extra por ejemplo:

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()
    for library in libraries:
        for book in library.books.filter(title__contains="Django"):
            print(book.name)
```

Lo siguiente se puede optimizar utilizando un objeto `Prefetch` , introducido en Django 1.7:

```
from django.db.models import Prefetch
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related(
        Prefetch('books', queryset=Book.objects.filter(title__contains="Django")
    ).all()
    for library in libraries:
        for book in library.books.all():
            print(book.name)  # Will print only books containing Django for each library
```

Reducir el número de consultas en el campo ForeignKey (problema n + 1)

Problema

Los querysets de Django se evalúan de manera perezosa. Por ejemplo:

```
# models.py:
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database
    books = Book.objects.all()
```

```
    for book in books:
        # Query the database on each iteration to get author (len(books) times)
        # if there is 100 books, there will have 100 queries plus the initial query
        book.author
        # ...

    # total : 101 queries
```

El código anterior hace que django consulte la base de datos del autor de cada libro. Esto es ineficiente, y es mejor tener solo una consulta.

Solución

Utilice `select_related` en `ForeignKey` si sabe que necesitará acceder más tarde a un campo `ForeignKey` .

```
# views.py
def myview(request):
    # Query the database.
    books = Books.objects.select_related('author').all()

    for book in books:
        # Does not query the database again, since `author` is pre-populated
        book.author
        # ...

    # total : 1 query
```

`select_related` también se puede utilizar en los campos de búsqueda:

```
# models.py:
class AuthorProfile(models.Model):
    city = models.CharField(max_length=100)

class Author(models.Model):
    name = models.CharField(max_length=100)
    profile = models.OneToOneField(AuthorProfile)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    books = Book.objects.select_related('author')\
                        .select_related('author__profile').all()

    for book in books:
        # Does not query database
        book.author.name
        # or
        book.author.profile.city
        # ...
```

```
    # total : 1 query
```

Obtener SQL para Django Queryset

El atributo de `query` en queryset le proporciona una sintaxis equivalente a SQL para su consulta.

```
>>> queryset = MyModel.objects.all()
>>> print(queryset.query)
SELECT "myapp_mymodel"."id", ... FROM "myapp_mymodel"
```

> **Advertencia:**
>
> Esta salida solo se debe utilizar para fines de depuración. La consulta generada no es específica del backend. Como tales, los parámetros no se citan correctamente, lo que lo hace vulnerable a la inyección de SQL, y es posible que la consulta ni siquiera sea ejecutable en su base de datos.

Obtener el primer y último registro de QuerySet

Para obtener el primer objeto:

```
MyModel.objects.first()
```

Para obtener los últimos objetos:

```
MyModel.objects.last()
```

Usando el objeto Filter First:

```
MyModel.objects.filter(name='simple').first()
```

Usando Filter Last objeto:

```
MyModel.objects.filter(name='simple').last()
```

Consultas avanzadas con objetos F

> Un objeto F () representa el valor de un campo de modelo o columna anotada. Permite hacer referencia a los valores de campo del modelo y realizar operaciones de base de datos usándolos sin tener que sacarlos de la base de datos a la memoria de Python. - F () expresiones

Es apropiado usar objetos `F()` siempre que necesite hacer referencia al valor de otro campo en su consulta. Por sí mismos, los objetos `F()` no significan nada, y no pueden y no deben llamarse fuera de un queryset. Se utilizan para hacer referencia al valor de un campo en el mismo queryset.

Por ejemplo, dado un modelo ...

```
SomeModel(models.Model):
    ...
    some_field = models.IntegerField()
```

... un usuario puede consultar objetos donde el valor de `some_field` es el doble de su `id` haciendo referencia al valor del campo de `id` mientras filtra usando `F()` esta manera:

```
SomeModel.objects.filter(some_field=F('id') * 2)
```

`F('id')` simplemente hace referencia al valor de `id` para esa misma instancia. Django lo usa para crear la declaración SQL correspondiente. En este caso algo muy parecido a esto:

```
SELECT * FROM some_app_some_model
WHERE some_field = ((id * 2))
```

Sin las expresiones `F()` , esto se lograría con SQL sin procesar o filtrado en Python (lo que reduce el rendimiento, especialmente cuando hay muchos objetos).

Referencias:

- Los filtros pueden hacer referencia a los campos en el modelo.
- Expresiones f
- Respuesta de TinyInstance

De la definición de la clase `F()` :

> Un objeto capaz de resolver referencias a objetos de consulta existentes. - fuente F

Nota: este ejemplo publicado provino de la respuesta que aparece arriba con el consentimiento de TinyInstance.

Lea Querysets en línea: https://riptutorial.com/es/django/topic/1235/querysets

Capítulo 44: RangeFields - un grupo de campos específicos de PostgreSQL

Sintaxis

- desde django.contrib.postgres.fields import * RangeField
- IntegerRangeField (** opciones)
- BigIntegerRangeField (** opciones)
- FloatRangeField (** opciones)
- DateTimeRangeField (** opciones)
- DateRangeField (** opciones)

Examples

Incluyendo campos de rango numérico en su modelo

Hay tres tipos de `RangeField` numérico en Python. `IntegerField` , `BigIntegerField` y `FloatField` . Se convierten a `psycopg2` `NumericRange` s, pero aceptan entradas como tuplas nativas de Python. **Se incluye el límite inferior y se excluye el límite superior.**

```
class Book(models.Model):
    name = CharField(max_length=200)
    ratings_range = IntegerRange()
```

Configurando para RangeField

1. agregue `'django.contrib.postgres'` a su `INSTALLED_APPS`
2. instalar `psycopg2`

Creando modelos con campos de rango numérico

Es más simple y más fácil ingresar valores como una tupla de Python en lugar de un `NumericRange` .

```
Book.objects.create(name='Pro Git', ratings_range=(5, 5))
```

Método alternativo con `NumericRange` :

```
Book.objects.create(name='Pro Git', ratings_range=NumericRange(5, 5))
```

Usando contiene

Esta consulta selecciona todos los libros con cualquier calificación inferior a tres.

```
bad_books = Books.objects.filter(ratings_range__contains=(1, 3))
```

Usando contenido_por

Esta consulta obtiene todos los libros con calificaciones mayores o iguales a cero y menores a seis.

```
all_books = Book.objects.filter(ratings_range_contained_by=(0, 6))
```

Usando superposición

Esta consulta obtiene todas las citas superpuestas de seis a diez.

```
Appointment.objects.filter(time_span__overlap=(6, 10))
```

Usando Ninguno para significar que no hay límite superior

Esta consulta selecciona todos los libros con una calificación mayor o igual a cuatro.

```
maybe_good_books = Books.objects.filter(ratings_range__contains=(4, None))
```

Rangos de operaciones

```
from datetime import timedelta

from django.utils import timezone
from psycopg2.extras import DateTimeTZRange

# To create a "period" object we will use psycopg2's DateTimeTZRange
# which takes the two datetime bounds as arguments
period_start = timezone.now()
period_end = period_start + timedelta(days=1, hours=3)
period = DateTimeTZRange(start, end)

# Say Event.timeslot is a DateTimeRangeField

# Events which cover at least the whole selected period,
Event.objects.filter(timeslot__contains=period)

# Events which start and end within selected period,
Event.objects.filter(timeslot__contained_by=period)

# Events which, at least partially, take place during the selected period.
Event.objects.filter(timeslot__overlap=period)
```

Lea RangeFields - un grupo de campos específicos de PostgreSQL en línea: https://riptutorial.com/es/django/topic/2630/rangefields---un-grupo-de-campos-especificos-de-postgresql

Capítulo 45: Relaciones de muchos a muchos

Examples

Con un modelo pasante.

```
class Skill(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Developer(models.Model):
    name = models.CharField(max_length=50)
    skills = models.ManyToManyField(Skill, through='DeveloperSkill')

class DeveloperSkill(models.Model):
    """Developer skills with respective ability and experience."""

    class Meta:
        order_with_respect_to = 'developer'
        """Sort skills per developer so that he can choose which
        skills to display on top for instance.
        """
        unique_together = [
            ('developer', 'skill'),
        ]
        """It's recommended that a together unique index be created on
        `(developer,skill)`. This is especially useful if your database is
        being access/modified from outside django. You will find that such an
        index is created by django when an explicit through model is not
        being used.
        """


    ABILITY_CHOICES = [
        (1, "Beginner"),
        (2, "Accustomed"),
        (3, "Intermediate"),
        (4, "Strong knowledge"),
        (5, "Expert"),
    ]

    developer = models.ForeignKey(Developer, models.CASCADE)
    skill = models.ForeignKey(Skill, models.CASCADE)
    """The many-to-many relation between both models is made by the
    above two foreign keys.

    Other fields (below) store information about the relation itself.
    """

    ability = models.PositiveSmallIntegerField(choices=ABILITY_CHOICES)
    experience = models.PositiveSmallIntegerField(help_text="Years of experience.")
```

Se recomienda que se cree un índice único en conjunto `(developer,skill)` . Esto es especialmente útil si se está accediendo / modificando su base de datos desde fuera de django. Encontrará que dicho índice es creado por django cuando no se está utilizando un modelo explícito a través de.

Simple muchos a muchos relación.

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Aquí definimos una relación donde un club tiene muchas `Person` y miembros y una Persona puede ser miembro de varios `Club` diferentes.

Aunque solo definimos dos modelos, django crea tres tablas en la base de datos para nosotros. Estas son `myapp_person` , `myapp_club` y myapp_club_members. Django crea automáticamente un índice único en las `myapp_club_members(club_id,person_id)` .

Uso de muchos campos de muchos

Usamos este modelo del primer ejemplo:

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Agrega a Tom y Bill al club nocturno:

```
tom = Person.objects.create(name="Tom", description="A nice guy")
bill = Person.objects.create(name="Bill", description="Good dancer")

nightclub = Club.objects.create(name="The Saturday Night Club")
nightclub.members.add(tom, bill)
```

Quien esta en el club

```
for person in nightclub.members.all():
    print(person.name)
```

Te regalaré

```
Tom
Bill
```

Lea Relaciones de muchos a muchos en línea:
https://riptutorial.com/es/django/topic/2379/relaciones-de-muchos-a-muchos

Capítulo 46: Seguridad

Examples

Protección de Cross Site Scripting (XSS)

Los ataques XSS consisten en inyectar código HTML (o JS) en una página. Consulte Qué es el script de sitios cruzados para obtener más información.

Para evitar este ataque, de forma predeterminada, Django escapa las cadenas que pasan a través de una variable de plantilla.

Dado el siguiente contexto:

```
context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': (
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&amp;quot; style=&amp;quot;font-size: 4000px">&amp;lt;script&amp;gt;alert(&amp;#39;hello
world!&amp;#39;);&amp;lt;/script&amp;gt;</p>
```

Si tiene variables que contienen HTML en las que confía y que realmente desea representar, debe decir explícitamente que es seguro:

```
<p class="{{ class_name|safe }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px">&amp;lt;script&amp;gt;alert(&amp;#39;hello
world!&amp;#39;);&amp;lt;/script&amp;gt;</p>
```

Si tiene un bloque que contiene varias variables que son seguras, puede deshabilitar el escape automático localmente:

```
{% autoescape off %}
<p class="{{ class_name }}">{{ paragraph }}</p>
{% endautoescape %}
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px"><script>alert('hello world!');</script></p>
```

También puede marcar una cadena como segura fuera de la plantilla:

```
from django.utils.safestring import mark_safe

context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': mark_safe(
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
```

```
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&amp;quot; style=&amp;quot;font-size: 4000px"><script>alert('hello
world!');</script></p>
```

Algunas utilidades de Django como `format_html` ya devuelven cadenas marcadas como seguras:

```
from django.utils.html import format_html

context = {
    'var': format_html('<b>{}</b> {}', 'hello', '<i>world!</i>'),
}
```

```
<p>{{ var }}</p>
<!-- Will be rendered as -->
<p><b>hello</b> &amp;lt;i&amp;gt;world!&amp;lt;/i&amp;gt;</p>
```

Protección de clickjacking

> Clickjacking es una técnica maliciosa de engañar a un usuario de la Web para que haga clic en algo diferente de lo que el usuario percibe que está haciendo clic.
> Aprende más

Para habilitar la protección de clickjacking, agregue `XFrameOptionsMiddleware` a sus clases de middleware. Esto ya debería estar allí si no lo eliminaste.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
]
```

Este middleware establece el encabezado 'X-Frame-Options' en todas sus respuestas, a menos que esté explícitamente exento o ya establecido (no se invalida si ya está establecido en la respuesta). De forma predeterminada, se establece en "SAMEORIGIN". Para cambiar esto, use la configuración `X_FRAME_OPTIONS` :

```
X_FRAME_OPTIONS = 'DENY'
```

Puede anular el comportamiento predeterminado por vista.

```
from django.utils.decorators import method_decorator
from django.views.decorators.clickjacking import (
    xframe_options_exempt, xframe_options_deny, xframe_options_sameorigin,
)

xframe_options_exempt_m = method_decorator(xframe_options_exempt, name='dispatch')
```

```
@xframe_options_sameorigin
def my_view(request, *args, **kwargs):
    """Forces 'X-Frame-Options: SAMEORIGIN'."""
    return HttpResponse(...)

@method_decorator(xframe_options_deny, name='dispatch')
class MyView(View):
    """Forces 'X-Frame-Options: DENY'."""

@xframe_options_exempt_m
class MyView(View):
    """Does not set 'X-Frame-Options' header when passing through the
    XFrameOptionsMiddleware.
    """
```

Protección de falsificación de solicitudes entre sitios (CSRF)

> La falsificación de solicitudes entre sitios, también conocida como ataque con un solo clic o sesión montada y abreviada como CSRF o XSRF, es un tipo de vulnerabilidad malintencionada de un sitio web donde se transmiten comandos no autorizados de un usuario en el que el sitio confía. Aprende más

Para habilitar la protección CSRF, agregue `CsrfViewMiddleware` a sus clases de middleware. Este middleware está habilitado por defecto.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.csrf.CsrfViewMiddleware',
    ...
]
```

Este middleware establecerá un token en una cookie en la respuesta saliente. Cuando una solicitud entrante utiliza un método inseguro (cualquier método excepto `GET` , `HEAD` , `OPTIONS` y `TRACE`), la cookie debe coincidir con un token que se envía como datos de formulario `csrfmiddlewaretoken` o como encabezado `X-CsrfToken` . Esto garantiza que el cliente que inicia la solicitud también es el propietario de la cookie y, por extensión, la sesión (autenticada).

Si se realiza una solicitud a través de `HTTPS` , se habilita la comprobación estricta de referencias. Si el encabezado `HTTP_REFERER` no coincide con el host de la solicitud actual o con un host en `CSRF_TRUSTED_ORIGINS` (nuevo en 1.9), se deniega la solicitud.

Los formularios que utilizan el método `POST` deben incluir el token CSRF en la plantilla. La etiqueta de plantilla `{% csrf_token %}` generará un campo oculto y garantizará que la cookie esté configurada en la respuesta:

```
<form method='POST'>
{% csrf_token %}
...
</form>
```

Las vistas individuales que no son vulnerables a los ataques CSRF pueden quedar exentas

utilizando el decorador `@csrf_exempt` :

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def my_view(request, *args, **kwargs):
    """Allows unsafe methods without CSRF protection"""
    return HttpResponse(...)
```

Aunque no se recomienda, puede deshabilitar `CsrfViewMiddleware` si muchas de sus vistas no son vulnerables a los ataques CSRF. En este caso, puede utilizar el decorador `@csrf_protect` para proteger las vistas individuales:

```
from django.views.decorators.csrf import csrf_protect

@csrf_protect
def my_view(request, *args, **kwargs):
    """This view is protected against CSRF attacks if the middleware is disabled"""
    return HttpResponse(...)
```

Lea Seguridad en línea: https://riptutorial.com/es/django/topic/2957/seguridad

Capítulo 47: Señales

Parámetros

Clase / Método	El porque
UserProfile () Clase	La clase UserProfile amplía el modelo de usuario predeterminado de Django .
método create_profile ()	El método create_profile () se ejecuta, siempre que se `post_save` señal `post_save` modelo de usuario de Django.

Observaciones

Ahora, los detalles.

Las señales de Django son una forma de informar a su aplicación de ciertas tareas (como un modelo antes o después de guardar o eliminar) cuando se lleva a cabo.

Estas señales le permiten realizar acciones de su elección inmediatamente después de que se libera la señal.

Por ejemplo, **cada vez que** se crea un nuevo Usuario de Django, el Modelo de Usuario emite una señal, con parámetros asociados como `sender=User` que le permite dirigir específicamente su escucha de señales a una actividad específica que ocurre, en este caso, una nueva creación de usuario .

En el ejemplo anterior, la intención es tener un objeto UserProfile creado, *inmediatamente* después de que se crea un objeto User. Por lo tanto, al escuchar una señal `post_save` del modelo de `User` (el modelo de usuario de Django predeterminado) específicamente, creamos un objeto `UserProfile` justo después de crear un nuevo `User` .

La documentación de Django proporciona una amplia documentación sobre todas las posibles señales disponibles .

Sin embargo, el ejemplo anterior es explicar en términos prácticos un caso de uso típico cuando se usan señales puede ser una adición útil.

"Con un gran poder viene una gran responsabilidad". Puede ser tentador tener señales dispersas en toda su aplicación o proyecto solo porque son increíbles. Bueno no lo hagas Porque son geniales, no los convierte en la solución de referencia para cada situación simple que se le ocurra.

Las señales son geniales para, como siempre, no todo. Login / Logouts, las señales son geniales. Modelos clave que liberan signos, como el Modelo de Usuario, si está bien.

La creación de señales para todos y cada uno de los modelos en su aplicación puede ser abrumadora en algún momento, y anular la idea general del uso de spjan de Django Signals.

No use señales cuando (según el libro Two Scoops of Django):

- La señal se relaciona con un modelo en particular y se puede mover a uno de los métodos de ese modelo, posiblemente llamado por `save()` .
- La señal se puede reemplazar con un método de administrador de modelos personalizado.
- La señal se relaciona con una vista particular y se puede mover a esa vista

Podría estar bien usar señales cuando:

- Su receptor de señal necesita realizar cambios en más de un modelo.
- Desea enviar la misma señal desde varias aplicaciones y hacer que se manejen de la misma manera por un receptor común.
- Desea invalidar un caché después de guardar un modelo.
- Tiene un escenario inusual que necesita una devolución de llamada, y no hay otra manera de manejarlo además de usar una señal. Por ejemplo, desea activar algo en función del `save()` o `init()` del modelo de una aplicación de terceros. No puede modificar el código de terceros y su extensión podría ser imposible, por lo que una señal proporciona un desencadenante para una devolución de llamada.

Examples

Ejemplo de extensión de perfil de usuario

Este ejemplo es un fragmento de código extraído del perfil de usuario de Django como un profesional.

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

Diferente sintaxis para publicar / pre una señal.

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver
```

```
class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

@receiver(post_save, sender=UserProfile)
def post_save_user(sender, **kwargs):
    user = kwargs.get('instance')
    if kwargs.get('created'):
        ...
```

Cómo encontrar si es una inserción o actualización en la señal de pre_save

Al utilizar `pre_save` , podemos determinar si una acción de `save` en nuestra base de datos fue sobre la actualización de un objeto existente o la creación de uno nuevo.

Para lograr esto, puede verificar el estado del objeto modelo:

```
    @receiver(pre_save, sender=User)
    def pre_save_user(sender, instance, **kwargs):
        if not instance._state.adding:
            print ('this is an update')
        else:
            print ('this is an insert')
```

Ahora, cada vez que se realiza una acción de `save` , la señal de `pre_save` se ejecutará e imprimirá:

- `this is an update` si la acción deriva de una acción de actualización.
- `this is an insert` si la acción deriva de una acción de inserción.

Tenga en cuenta que este método no requiere ninguna consulta de base de datos adicional.

Heredando señales en modelos extendidos

Las señales de Django están restringidas a firmas de clase precisas al registrarse, y por lo tanto, los modelos subclasificados no se registran inmediatamente en la misma señal.

Toma este modelo y señal por ejemplo

```
class Event(models.Model):
    user = models.ForeignKey(User)


class StatusChange(Event):
    ...


class Comment(Event):
    ...


def send_activity_notification(sender, instance: Event, raw: bool, **kwargs):
```

```
    """
    Fire a notification upon saving an event
    """

    if not raw:
        msg_factory = MessageFactory(instance.id)
        msg_factory.on_activity(str(instance))
post_save.connect(send_activity_notification, Event)
```

Con los modelos extendidos, debe adjuntar manualmente la señal en cada subclase, de lo contrario no se verán afectados.

```
post_save.connect(send_activity_notification, StatusChange)
post_save.connect(send_activity_notification, Comment)
```

Con Python 3.6, puede aprovechar algunos métodos de clase adicionales construidos en clases para automatizar este enlace.

```
class Event(models.Model):

    @classmethod
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        post_save.connect(send_activity_notification, cls)
```

Lea Señales en línea: https://riptutorial.com/es/django/topic/2555/senales

Capítulo 48: Tareas asíncronas (Apio)

Observaciones

El apio es una cola de tareas que puede ejecutar trabajos en segundo plano o programados y se integra bastante bien con Django. El apio requiere algo conocido como **intermediario** de mensajes para pasar mensajes de invocación a los trabajadores. Este intermediario de mensajes puede ser redis, rabbitmq o incluso Django ORM / db, aunque no es un enfoque recomendado.

Antes de comenzar con el ejemplo, deberá configurar el apio. Para configurar el apio, cree un archivo `celery_config.py` en la aplicación principal, paralelo al archivo `settings.py` .

```
from __future__ import absolute_import
import os
from celery import Celery
from django.conf import settings

# broker url
BROKER_URL = 'redis://localhost:6379/0'

# Indicate Celery to use the default Django settings module
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

app = Celery('config')
app.config_from_object('django.conf:settings')
# if you do not need to keep track of results, this can be turned off
app.conf.update(
    CELERY_RESULT_BACKEND=BROKER_URL,
)

# This line will tell Celery to autodiscover all your tasks.py that are in your app folders
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

Y en el archivo `__init__.py` la aplicación principal, importe la aplicación de apio. Me gusta esto

```
# -*- coding: utf-8 -*-
# Not required for Python 3.
from __future__ import absolute_import

from .celery_config import app as celery_app  # noqa
```

Para ejecutar el trabajo de apio, use este comando en el nivel donde se encuentra manage.py.

```
# pros is your django project,
celery -A proj worker -l info
```

Examples

Ejemplo simple para sumar 2 números.

Para empezar:

1. Instalar apio `pip install celery`
2. configurar apio (diríjase a la sección de comentarios)

```
from __future__ import absolute_import, unicode_literals

from celery.decorators import task


@task
def add_number(x, y):
    return x + y
```

Puede ejecutar esto de forma asincrónica utilizando el método `.delay()` .

`add_number.delay(5, 10)` , donde 5 y 10 son los argumentos para la función `add_number`

Para verificar si la función asíncrona ha finalizado la operación, puede usar la función `.ready()` en el objeto asíncrono devuelto por el método de `delay` .

Para obtener el resultado del cálculo, puede usar el atributo `.result` en el objeto asíncrono.

Ejemplo

```
async_result_object = add_number.delay(5, 10)
if async_result_object.ready():
    print(async_result_object.result)
```

Lea Tareas asíncronas (Apio) en línea: https://riptutorial.com/es/django/topic/5481/tareas-asincronas--apio-

Capítulo 49: Transacciones de base de datos

Examples

Transacciones atómicas

Problema

Por defecto, Django confirma inmediatamente los cambios en la base de datos. Cuando se producen excepciones durante una serie de confirmaciones, esto puede dejar su base de datos en un estado no deseado:

```
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

En el siguiente escenario:

```
>>> create_category('clothing', ['shirt', 'trousers', 'tie'])
---------------------------------------------------------------------------
ValueError: Product 'trousers' already exists
```

Se produce una excepción al intentar agregar el producto de pantalón a la categoría de ropa. En este punto, la categoría en sí ya se ha agregado, y el producto de la camisa se ha agregado a ella.

La categoría incompleta y el producto que contiene deberían eliminarse manualmente antes de corregir el código y llamar al método `create_category()` una vez más, ya que de lo contrario se crearía una categoría duplicada.

Solución

El módulo `django.db.transaction` permite combinar múltiples cambios de base de datos en una transacción atómica :

> [una] serie de operaciones de base de datos de manera que ocurran todas o no ocurra nada.

Aplicado al escenario anterior, esto se puede aplicar como decorador :

```
from django.db import transaction

@transaction.atomic
```

```
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

O utilizando un administrador de contexto :

```
def create_category(name, products):
    with transaction.atomic():
        category = Category.objects.create(name=name)
        product_api.add_products_to_category(category, products)
        activate_category(category)
```

Ahora, si se produce una excepción en cualquier etapa de la transacción, no se confirmarán cambios en la base de datos.

Lea Transacciones de base de datos en línea:
https://riptutorial.com/es/django/topic/5555/transacciones-de-base-de-datos

Capítulo 50: Usando Redis con Django - Caching Backend

Observaciones

El uso de django-redis-cache o django-redis son soluciones efectivas para almacenar todos los elementos almacenados en caché. Si bien es posible que Redis se configure directamente como `SESSION_ENGINE` , una estrategia efectiva es configurar el almacenamiento en caché (como se indica arriba) y declarar su caché predeterminado como `SESSION_ENGINE` . Si bien este es realmente el tema de otro artículo de documentación, su relevancia lleva a la inclusión.

Simplemente agregue lo siguiente a `settings.py` :

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Examples

Usando django-redis-cache

Una implementación potencial de Redis como una utilidad de almacenamiento en caché de fondo es el paquete django-redis-cache .

Este ejemplo asume que ya tiene un servidor Redis operativo .

```
$ pip install django-redis-cache
```

Edite su `settings.py` para incluir un objeto `CACHES` (consulte la documentación de Django sobre almacenamiento en caché).

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.RedisCache',
        'LOCATION': 'localhost:6379',
        'OPTIONS': {
            'DB': 0,
        }
    }
}
```

Utilizando django-redis

Una implementación potencial de Redis como una utilidad de almacenamiento en caché de fondo es el paquete django-redis .

Este ejemplo asume que ya tiene un servidor Redis operativo .

```
$ pip install django-redis
```

Edite su `settings.py` para incluir un objeto `CACHES` (consulte la documentación de Django sobre almacenamiento en caché).

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

Lea Usando Redis con Django - Caching Backend en línea:
https://riptutorial.com/es/django/topic/4085/usando-redis-con-django---caching-backend

Capítulo 51: Vistas basadas en clase

Observaciones

Cuando usamos CBV, a menudo necesitamos saber exactamente qué métodos podemos sobrescribir para cada clase genérica. Esta página de la documentación de django enumera todas las clases genéricas con todos sus métodos simplificados y los atributos de clase que podemos usar.

Además, el sitio web Classy Class Based View proporciona la misma información con una interfaz interactiva agradable.

Examples

Vistas basadas en clase

Las vistas basadas en clase le permiten concentrarse en lo que hace que sus vistas sean especiales.

Una página estática sobre la página puede no tener nada especial, excepto la plantilla utilizada. Utilice un TemplateView ! Todo lo que tienes que hacer es establecer un nombre de plantilla. Trabajo hecho. Siguiente.

vistas.py

```
from django.views.generic import TemplateView


class AboutView(TemplateView):
    template_name = "about.html"
```

urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url('^about/', views.AboutView.as_view(), name='about'),
]
```

Observe cómo no utilizamos directamente `AboutView` en la url. Esto se debe a que se espera una llamada y eso es exactamente lo que `as_view()` devuelve.

Datos de contexto

A veces, su plantilla necesita un poco más de información. Por ejemplo, nos gustaría tener al usuario en el encabezado de la página, con un enlace a su perfil junto al enlace de cierre de sesión. En estos casos, utilice el método `get_context_data` .

vistas.py

```
class BookView(DetailView):
    template_name = "book.html"

    def get_context_data(self, **kwargs)
        """ get_context_data let you fill the template context """
        context = super(BookView, self).get_context_data(**kwargs)
        # Get Related publishers
        context['publishers'] = self.object.publishers.filter(is_active=True)
        return context
```

Debe llamar al método get_context_data en la superclase y devolverá la instancia de contexto predeterminada. Cualquier elemento que agregue a este diccionario estará disponible para la plantilla.

libro.html

```
<h3>Active publishers</h3>
<ul>
  {% for publisher in publishers %}
    <li>{{ publisher.name }}</li>
  {% endfor %}
</ul>
```

Vistas de lista y detalles

Las vistas de plantilla están bien para la página estática y podría usarlas para todo con `get_context_data` pero sería apenas mejor que usar la función como vistas.

Ingrese a ListView y DetailView

app / models.py

```
from django.db import models

class Pokemon(models.Model):
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)
    slug = models.CharField(max_length=48)
```

app / views.py

```
from django.views.generic import ListView, DetailView
from .models import Pokemon


class PokedexView(ListView):
    """ Provide a list of Pokemon objects """
    model = Pokemon
    paginate_by = 25

class PokemonView(DetailView):
    model = Pokemon
```

Eso es todo lo que necesita para generar una vista que enumere todos sus objetos de modelos y vistas de un elemento singular. La lista está incluso paginada. Puede proporcionar `template_name` si desea algo específico. Por defecto, se genera a partir del nombre del modelo.

app / templates / app / pokemon_list.html

```
<!DOCTYPE html>
<title>Pokedex</title>
<ul>{% for pokemon in pokemon_list %}
    <li><a href="{% url "app:pokemon" pokemon.pk %}">{{ pokemon.name }}</a>
        &ndash; {{ pokemon.species }}
</ul>
```

El contexto se rellena con la lista de objetos bajo dos nombres, `object_list` y una segunda compilación a partir del nombre del modelo, aquí `pokemon_list` . Si ha paginado la lista, también debe cuidar el enlace anterior y el siguiente. El objeto Paginator puede ayudar con eso, también está disponible en los datos de contexto.

app / templates / app / pokemon_detail.html

```
<!DOCTYPE html>
<title>Pokemon {{ pokemon.name }}</title>
<h1>{{ pokemon.name }}</h1>
<h2>{{ pokemon.species }} </h2>
```

Como antes, el contexto se completa con su objeto modelo bajo el nombre de `object` y `pokemon` , el segundo se deriva del nombre del modelo.

app / urls.py

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

```
from . import views

app_name = 'app'
urlpatterns = [
    url(r'^pokemon/$', views.PokedexView.as_view(), name='pokedex'),
    url(r'^pokemon/(?P<pk>\d+)/$', views.PokemonView.as_view(), name='pokemon'),
]
```

En este fragmento, la url para la vista de detalle se crea utilizando la clave principal. También es posible usar una bala como argumento. Esto da una url más bonita que es más fácil de recordar. Sin embargo, requiere la presencia de un campo llamado slug en su modelo.

```
url(r'^pokemon/(?P<slug>[A-Za-z0-9_-]+)/$', views.PokemonView.as_view(), name='pokemon'),
```

Si un campo llamado `slug` no está presente, se puede utilizar el `slug_field` puesta en `DetailView` para que apunte a un campo diferente.

Para la paginación, use una página obtener parámetros o poner una página directamente en la url.

Creación de forma y objeto.

Escribir una vista para crear un objeto puede ser bastante aburrido. Debe mostrar un formulario, debe validarlo, debe guardar el elemento o devolver el formulario con un error. A menos que uses una de las vistas de edición genéricas .

app / views.py

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Pokemon


class PokemonCreate(CreateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonUpdate(UpdateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonDelete(DeleteView):
    model = Pokemon
    success_url = reverse_lazy('pokedex')
```

`CreateView` y `UpdateView` tienen dos atributos, `model` y `fields` requeridos. De forma predeterminada, ambos usan un nombre de plantilla basado en el nombre del modelo con el sufijo '_form'. Solo puede cambiar el sufijo con el atributo template_name_suffix. El DeleteView muestra un mensaje de confirmación antes de eliminar el objeto.

Tanto `UpdateView` como `DeleteView` deben `DeleteView` en el objeto. Utilizan el mismo método que `DetailView` , extraen la variable de la URL y hacen coincidir los campos del objeto.

app / templates / app / pokemon_form.html (extraer)

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

`form` contiene el formulario con todos los campos necesarios. Aquí, se mostrará con un párrafo para cada campo debido a `as_p` .

app / templates / app / pokemon_confirm_delete.html (extraer)

```
<form action="" method="post">
    {% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input type="submit" value="Confirm" />
</form>
```

La etiqueta `csrf_token` es necesaria debido a la protección de django contra la falsificación de solicitudes. La acción del atributo se deja vacía, ya que la URL que muestra el formulario es la misma que la que maneja la eliminación / guardar.

Dos problemas permanecen con el modelo, si se usa lo mismo que con la lista y el ejemplo de detalle. En primer lugar, crear y actualizar se quejará de una URL de redireccionamiento que falta. Eso se puede resolver agregando un `get_absolute_url` al modelo pokemon. El segundo problema es que la confirmación de eliminación no muestra información significativa. Para resolver esto, la solución más sencilla es agregar una representación de cadena.

app / models.py

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible


@python_2_unicode_compatible
class Pokemon(models.Model):
    name = models.CharField(max_length=24)
```

```
    species = models.CharField(max_length=48)

    def get_absolute_url(self):
        return reverse('app:pokemon', kwargs={'pk':self.pk})

    def __str__(self):
        return self.name
```

El decorador de la clase se asegurará de que todo funcione sin problemas bajo python 2.

Ejemplo minimo

views.py :

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

urls.py :

```
from django.conf.urls import url
from myapp.views import MyView

urlpatterns = [
    url(r'^about/$', MyView.as_view()),
]
```

Aprenda más sobre la documentación de Django »

Vistas basadas en la clase de Django: Ejemplo de CreateView

Con las vistas genéricas basadas en clase, es muy simple y fácil crear las vistas CRUD desde nuestros modelos. A menudo, el administrador integrado de Django no es suficiente o no es el preferido y necesitamos rodar nuestras propias vistas de CRUD. Los CBV pueden ser muy útiles en tales casos.

La clase `CreateView` necesita 3 cosas: un modelo, los campos para usar y la url de éxito.

Ejemplo:

```
from django.views.generic import CreateView
from .models import Campaign

class CampaignCreateView(CreateView):
    model = Campaign
    fields = ('title', 'description')

    success_url = "/campaigns/list"
```

Una vez que la creación es exitosa, el usuario es redirigido a `success_url` . También podemos definir un método `get_success_url` en `get_success_url` lugar y usar `reverse` o `reverse_lazy` para obtener la url de éxito.

Ahora, necesitamos crear una plantilla para esta vista. La plantilla debe tener un nombre en el formato `<app name>/<model name>_form.html` . El nombre del modelo debe estar en mayúsculas inferiores. Por ejemplo, si el nombre de mi aplicación es `dashboard` , entonces para la vista de creación anterior, debo crear una plantilla llamada `dashboard/campaign_form.html` .

En la plantilla, una variable de `form` contendría el formulario. Aquí hay un código de ejemplo para la plantilla:

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

Ahora es el momento de agregar la vista a nuestros patrones de url.

```
url('^campaign/new/$', CampaignCreateView.as_view(), name='campaign_new'),
```

Si visitamos la URL, deberíamos ver un formulario con los campos que elegimos. Cuando lo enviemos, intentará crear una nueva instancia del modelo con los datos y guardarlo. En caso de éxito, el usuario será redirigido a la URL de éxito. En caso de errores, el formulario se mostrará de nuevo con los mensajes de error.

Una vista, formas múltiples

Este es un ejemplo rápido del uso de múltiples formularios en una vista de Django.

```
from django.contrib import messages
from django.views.generic import TemplateView

from .forms import AddPostForm, AddCommentForm
from .models import Comment

class AddCommentView(TemplateView):

    post_form_class = AddPostForm
    comment_form_class = AddCommentForm
    template_name = 'blog/post.html'

    def post(self, request):
        post_data = request.POST or None
        post_form = self.post_form_class(post_data, prefix='post')
        comment_form = self.comment_form_class(post_data, prefix='comment')

        context = self.get_context_data(post_form=post_form,
                                        comment_form=comment_form)

        if post_form.is_valid():
            self.form_save(post_form)
```

```
        if comment_form.is_valid():
            self.form_save(comment_form)

        return self.render_to_response(context)

    def form_save(self, form):
        obj = form.save()
        messages.success(self.request, "{} saved successfully".format(obj))
        return obj

    def get(self, request, *args, **kwargs):
        return self.post(request, *args, **kwargs)
```

Lea Vistas basadas en clase en línea: https://riptutorial.com/es/django/topic/1220/vistas-basadas-en-clase

Capítulo 52: Vistas genéricas

Introducción

Las vistas genéricas son vistas que realizan una determinada acción predefinida, como crear, editar o eliminar objetos, o simplemente mostrar una plantilla.

Las vistas genéricas deben distinguirse de las vistas funcionales, que siempre se escriben a mano para realizar las tareas requeridas. En pocas palabras, se puede decir que las vistas genéricas deben configurarse, mientras que las vistas funcionales deben programarse.

Las vistas genéricas pueden ahorrar mucho tiempo, especialmente cuando tiene que realizar muchas tareas estandarizadas.

Observaciones

Estos ejemplos muestran que las vistas genéricas generalmente hacen que las tareas estandarizadas sean mucho más simples. En lugar de programar todo desde cero, configura lo que otras personas ya han programado para usted. Esto tiene sentido en muchas situaciones, ya que le permite concentrarse más en el diseño de sus proyectos que en los procesos en segundo plano.

Entonces, ¿ *siempre* deberías usarlos? No. Solo tienen sentido siempre que sus tareas sean bastante estandarizadas (cargar, editar, eliminar objetos) y cuanto más repetitivas sean sus tareas. Usar una vista genérica específica solo una vez y luego anular todos sus métodos para realizar tareas muy específicas puede que no tenga sentido. Puede que estés mejor con una vista funcional aquí.

Sin embargo, si tiene muchas vistas que requieren esta funcionalidad o si sus tareas coinciden exactamente con las tareas definidas de una vista genérica específica, entonces las vistas genéricas son exactamente lo que necesita para simplificar su vida.

Examples

Ejemplo Mínimo: Funcional vs. Vistas Genéricas

Ejemplo para una vista funcional para crear un objeto. Excluyendo comentarios y líneas en blanco, necesitamos 15 líneas de código:

```
# imports
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect

from .models import SampleObject
from .forms import SampleObjectForm
```

```
# view functioon
def create_object(request):

    # when request method is 'GET', show the template
    if request.method == GET:
        # perform actions, such as loading a model form
        form = SampleObjectForm()
        return render_to_response('template.html', locals())

    # if request method is 'POST', create the object and redirect
    if request.method == POST:
        form = SampleObjectForm(request.POST)

        # save object and redirect to success page if form is valid
        if form.is_valid:
            form.save()
            return HttpResponseRedirect('url_to_redirect_to')

        # load template with form and show errors
        else:
            return render_to_response('template.html', locals())
```

Ejemplo para una 'Vista genérica basada en clase' para realizar la misma tarea. Solo necesitamos 7 líneas de código para lograr la misma tarea:

```
from django.views.generic import CreateView

from .models import SampleObject
from .forms import SampleObjectForm

class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'
```

Personalizar vistas genéricas

El ejemplo anterior solo funciona si sus tareas son tareas completamente estándar. No agrega contexto adicional aquí, por ejemplo.

Hagamos un ejemplo más realista. Supongamos que queremos agregar un título de página a la plantilla. En la vista funcional, esto funcionaría así: con solo una línea adicional:

```
def create_object(request):
    page_title = 'My Page Title'

    # ...

    return render_to_response('template.html', locals())
```

Esto es más difícil (o contra-intuitivo) de lograr con vistas genéricas. Como están basados en la clase, debe anular uno o varios de los métodos de la clase para lograr el resultado deseado. En nuestro ejemplo, necesitamos anular el método *get_context_data de* la clase de la *siguiente* manera:

```
class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context
```

Aquí, necesitamos cuatro líneas adicionales para codificar en lugar de solo una, al menos para la *primera* variable de contexto adicional que deseamos agregar.

Vistas genéricas con mixins

El verdadero poder de las vistas genéricas se despliega cuando las combinas con Mixins. Un mixin es simplemente otra clase definida por usted cuyos métodos pueden ser heredados por su clase de vista.

Supongamos que desea que cada vista muestre la variable adicional 'page_title' en la plantilla. En lugar de anular el método get_context_data cada vez que defina la vista, cree un mixin con este método y deje que sus vistas se hereden de este mixin. Suena más complicado de lo que realmente es:

```
# Your Mixin
class CustomMixin(object):

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context

# Your view function now inherits from the Mixin
class CreateObject(CustomMixin, CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'

# As all other view functions which need these methods
class EditObject(CustomMixin, EditView):
    model = SampleObject
    # ...
```

La belleza de esto es que su código se vuelve mucho más estructurado de lo que es en su mayoría con vistas funcionales. Toda su lógica detrás de tareas específicas se encuentra en un solo lugar y en un solo lugar. Además, ahorrará enormes cantidades de tiempo, especialmente cuando tiene muchas vistas que siempre realizan las mismas tareas, excepto con diferentes objetos.

Lea Vistas genéricas en línea: https://riptutorial.com/es/django/topic/9452/vistas-genericas

Capítulo 53: Zonas horarias

Introducción

Las zonas horarias son a menudo una molestia para los desarrolladores. Django ofrece algunas excelentes utilidades a su disposición para facilitar el trabajo con las zonas horarias.

Incluso si su proyecto está operando en una sola zona horaria, sigue siendo una buena práctica almacenar datos como UTC en su base de datos para manejar los casos de ahorro de luz diurna. Si está operando en múltiples zonas horarias, almacenar datos de tiempo como UTC es una necesidad.

Examples

Habilitar soporte de zona horaria

Primero es primero, asegúrese de que `USE_TZ = True` en su archivo `settings.py` . También establezca un valor de zona horaria predeterminado en `TIME_ZONE` , como `TIME_ZONE='UTC'` . Vea una lista completa de zonas horarias aquí .

Si `USE_TZ` es Falso, `TIME_ZONE` será la zona horaria que Django usará para almacenar todas las fechas de los datos. Cuando `USE_TZ` está habilitado, `TIME_ZONE` es la zona horaria predeterminada que Django usará para mostrar los tiempos de las fechas en las plantillas e interpretar los tiempos de los datos ingresados en los formularios.

Con el soporte de zona horaria habilitado, django almacenará los datos de `datetime` y `datetime` en la base de datos como la zona horaria `UTC`

Configuración de zonas horarias de sesión

Los objetos `datetime.datetime` de Python tienen un atributo `tzinfo` que se utiliza para almacenar información de zona horaria. Cuando se establece el atributo, el objeto se considera consciente, cuando el atributo no se establece, se considera un ingenuo.

Para asegurarse de que una zona horaria sea ingenua o consciente, puede usar `.is_naive()` y `.is_aware()`

Si tiene `USE_TZ` habilitado en su archivo `settings.py` , una `datetime` y `datetime` tendrá información de zona horaria adjunta siempre que su `TIME_ZONE` predeterminado esté configurado en `settings.py`

Si bien esta zona horaria predeterminada puede ser buena en algunos casos, es probable que no sea suficiente, especialmente si está manejando usuarios en múltiples zonas horarias. Para lograr esto, se debe utilizar middleware.

```
import pytz
```

```
from django.utils import timezone

# make sure you add `TimezoneMiddleware` appropriately in settings.py
class TimezoneMiddleware(object):
    """
    Middleware to properly handle the users timezone
    """

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # make sure they are authenticated so we know we have their tz info.
        if request.user.is_authenticated():
            # we are getting the users timezone that in this case is stored in
            # a user's profile
            tz_str = request.user.profile.timezone
            timezone.activate(pytz.timezone(tz_str))
        # otherwise deactivate and the default time zone will be used anyway
        else:
            timezone.deactivate()

        response = self.get_response(request)
        return response
```

Hay algunas cosas nuevas que están sucediendo. Para obtener más información sobre el middleware y lo que hace, consulte esa parte de la documentación . En `__call__` estamos manejando la configuración de los datos de la zona horaria. Al principio, nos aseguramos de que el usuario esté autenticado, para asegurarnos de que tenemos datos de la zona horaria para este usuario. Una vez que sabemos que lo hacemos, activamos la zona horaria para la sesión de los usuarios utilizando `timezone.activate()` . Para convertir la cadena de zona horaria que tenemos a algo utilizable por datetime, usamos `pytz.timezone(str)` .

Ahora, cuando se accede a los objetos de fecha y hora en plantillas, se convertirán automáticamente del formato 'UTC' de la base de datos a cualquier zona horaria en la que se encuentre el usuario. Simplemente acceda al objeto de fecha y hora y se establecerá su zona horaria, asumiendo que el middleware anterior esté configurado correctamente.

```
{{ my_datetime_value }}
```

Si desea un control detallado sobre si se usa la zona horaria del usuario, mire lo siguiente:

```
{% load tz %}
{% localtime on %}
    {# this time will be respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}

{% localtime off %}
    {# this will not respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}
```

Tenga en cuenta que este método descrito solo funciona en Django 1.10 y en. Para admitir

django desde antes de 1.10, busque en MiddlewareMixin

Lea Zonas horarias en línea: https://riptutorial.com/es/django/topic/10566/zonas-horarias

Creditos

S. No	Capítulos	Contributors
1	Empezando con Django	A. Raza, Abhishek Jain, Aidas Bendoraitis, Alexander Tyapkov, Ankur Gupta, Anthony Pham, Antoine Pinsard, arifin4web, Community, e4c5, elbear, ericdwang, ettanany, Franck Dernoncourt, greatwolf, ilse2005, Ivan Semochkin, J F, Jared Hooper, John, John Moutafis, JRodDynamite, Kid Binary, knbk, Louis, Luis Alberto Santana, lxer, maciek, McAbra, MiniGunnR, mnoronha, Nathan Osman, naveen.panwar, nhydock, Nikita Davidenko, noɥʎdʎzɐɔ, Rahul Gupta, rajarshig, Ron, ruddra, sarvajeetsuman, shacker, ssice, Stryker, techydesigner, The Brewmaster, Thereissoupinmyfly, Tom, WesleyJohnson, Zags
2	¿Cómo usar Django con Cookiecutter?	Atul Mishra, noɥʎdʎzɐɔ, OliPro007, RamenChef
3	Administración	Antoine Pinsard, coffee-grinder, George H., Ivan Semochkin, noɥʎdʎzɐɔ, ssice
4	Agregaciones de modelos	Ian Clark, John Moutafis, ravigadila
5	Ajustes	allo, Antoine Pinsard, Brian Artschwager, fredley, J F, knbk, Louis, Louis Barranqueiro, lxer, Maxime Lorant, NBajanca, Nils Werner, ProfSmiles, RamenChef, Sanyam Khurana, Sayse, Selcuk, SpiXel, ssice, sudshekhar, Tema, The Brewmaster
6	ArrayField - un campo específico de PostgreSQL	Antoine Pinsard, e4c5, noɥʎdʎzɐɔ
7	Backends de autenticación	knbk, Rahul Gupta
8	Comandos de gestión	Antoine Pinsard, aquasan, Brian Artschwager, HorsePunchKid, Ivan Semochkin, John Moutafis, knbk, lxer, MarZab, Nikolay Fominyh, pbaranay, ptim, Rana Ahmed, techydesigner, Zags
9	Cómo restablecer las migraciones django	Cristus Cleetus
10	Configuración de la base de datos	Ahmad Anwar, Antoine Pinsard, Evans Murithi, Kid Binary, knbk , lxer, Majid, Peter Mortensen

11	CRUD en Django	aisflat439, George H.
12	Depuración	Antoine Pinsard, Ashutosh, e4c5, Kid Binary, knbk, Sayse, Udi
13	Despliegue	Antoine Pinsard, Arpit Solanki, CodeFanatic23, I Am Batman, Ivan Semochkin, knbk, lxer, Maxime S., MaxLunar, Meska, no ɥʎdʎzɐɔ, rajarshig, Rishabh Agrahari, Roald Nefs, Rohini Choudhary, sebb
14	Django desde la línea de comandos.	e4c5, OliPro007
15	Django Rest Framework	The Brewmaster
16	Django y redes sociales	Aidas Bendoraitis, aisflat439, Carlos Rojas, Ivan Semochkin, Rexford, Simplans
17	Ejecución de apio con supervisor	RéÑjïth, sebb
18	Enrutadores de base de datos	fredley, knbk
19	Enrutamiento de URL	knbk
20	Estructura del proyecto	Antoine Pinsard, naveen.panwar, nicorellius
21	Etiquetas de plantillas y filtros	Antoine Pinsard, irakli khitarishvili, knbk, Medorator, naveen.panwar, The_Cthulhu_Kid
22	Examen de la unidad	Adrian17, Antoine Pinsard, e4c5, Kim, Matthew Schinckel, Maxime Lorant, Patrik Stenmark, SandroM, sudshekhar, Zags
23	Explotación florestal	Antwane, Brian Artschwager, RamenChef
24	Extendiendo o Sustituyendo Modelo de Usuario	Antoine Pinsard, Jon Clements, mnoronha, Raito, Rexford, rigdonmr, Rishabh Agrahari, Roald Nefs, techydesigner, The_Cthulhu_Kid
25	F () expresiones	Antoine Pinsard, John Moutafis, Linville, Omar Shehata, RamenChef, Roald Nefs
26	filtro django	4444, Ahmed Atalla
27	Form Widgets	Antoine Pinsard, ettanany
28	Formas	Aidas Bendoraitis, Antoine Pinsard, Daniel Rucci, ettanany,

		George H., knbk, NBajanca, nicorellius, RamenChef, rumman0786, sudshekhar, trpt4him
29	Formsets	naveen.panwar
30	Gestores personalizados y Querysets	abidibo, knbk, sudshekhar, Trivial
31	Integración continua con Jenkins	pnovotnak
32	Internacionalización	Antoine Pinsard, dmvrtx
33	JSONField - un campo específico de PostgreSQL	Antoine Pinsard, Daniil Ryzhkov, Matthew Schinckel, noɥʎdʎzɐɔ, Omar Shehata, techydesigner
34	Mapeo de cadenas a cadenas con HStoreField - un campo específico de PostgreSQL	noɥʎdʎzɐɔ
35	Meta: Pautas de documentación.	Antoine Pinsard
36	Middleware	AlvaroAV, Antoine Pinsard, George H., knbk, lxer, nhydock, Omar Shehata, Peter Mortensen, Trivial, William Reed
37	Migraciones	Antoine Pinsard, engineercoding, Joey Wilhelm, knbk, MicroPyramid, ravigadila, Roald Nefs
38	Modelo de referencia de campo	Burhan Khalid, Husain Basrawala, knbk, Matt Seymour, Rod Xavier, scriptmonster, techydesigner, The_Cthulhu_Kid
39	Modelos	Aidas Bendoraitis, Alireza Aghamohammadi, alonisser, Antoine Pinsard, aquasan, Arpit Solanki, atomh33ls, coffee-grinder, DataSwede, ettanany, Gahan, George H., gkr, Ivan Semochkin, Jamie Cockburn, Joey Wilhelm, kcrk, knbk, Linville, lxer, maazza, Matt Seymour, MuYi, Navid777, nhydock, noɥʎdʎzɐɔ , pbaranay, PhoebeB, Rana Ahmed, Saksow, Sanyam Khurana, scriptmonster, Selcuk, SpiXel, sudshekhar, techydesigner, The_Cthulhu_Kid, Utsav T, waterproof, zurfyx
40	Plantilla	Adam Starrh, Alasdair, Aniket, Antoine Pinsard, Brian H., coffee-grinder, doctorsherlock, fredley, George H., gkr, lxer, Stephen Leppik, Zags

41	Procesadores de contexto	Antoine Pinsard, Brian Artschwager, Dan Russell, Daniil Ryzhkov, fredley
42	Puntos de vista	ettanany, HorsePunchKid, John Moutafis
43	Querysets	Antoine Pinsard, Brian Artschwager, Chalist, coffee-grinder, DataSwede, e4c5, Evans Murithi, George H., John Moutafis, Justin, knbk, Louis Barranqueiro, Maxime Lorant, MicroPyramid, nima, ravigadila, Sanyam Khurana, The Brewmaster
44	RangeFields - un grupo de campos específicos de PostgreSQL	Antoine Pinsard, noɥʎdʎzɐƆ
45	Relaciones de muchos a muchos	Antoine Pinsard, e4c5, knbk, Kostronor
46	Seguridad	Antoine Pinsard, knbk
47	Señales	Antoine Pinsard, e4c5, Hetdev, John Moutafis, Majid, nhydock, Rexford
48	Tareas asíncronas (Apio)	iankit, Mevin Babu
49	Transacciones de base de datos	Ian Clark
50	Usando Redis con Django - Caching Backend	Majid, The Brewmaster
51	Vistas basadas en clase	Antoine Pinsard, Antwane, coffee-grinder, e4c5, gkr, knbk, maciek, masnun, Maxime Lorant, nicorellius, pleasedontbelong, Pureferret
52	Vistas genéricas	nikolas-berlin
53	Zonas horarias	William Reed