Contenidos a Trabajar

1. Métodos del Modelo
2. Introducción a las vistas basadas en clases
   - Usando vistas basadas en clases
   - Usando mezclas (Mixins)
   - Manejo de Formularios
3. Más sobre vistas basadas en clases
   - Subclasificación de vistas genéricas

Agencia de Aprendizaje a lo largo de la vida

Métodos del Modelo

Defina métodos personalizados en un modelo para agregar funciones personalizadas de "nivel de registro" a sus objetos. Mientras que los métodos del **Manager** están destinados a hacer cosas "en toda la tabla", los métodos modelo deben actuar en una instancia de modelo particular.

Esta es una técnica valiosa para mantener la lógica empresarial en un solo lugar: el modelo.Por ejemplo, este modelo tiene algunos métodos personalizados:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

def baby_boomer_status(self):
    "Returns the person's baby-boomer status."
    import datetime
    if self.birth_date < datetime.date(1945, 8, 1):
        return "Pre-boomer"
    elif self.birth_date < datetime.date(1965, 1, 1):
        return "Baby boomer"
    else:
        return "Post-boomer"

@property
def full_name(self):
    "Returns the person's full name."
    return '%s %s' % (self.first_name, self.last_name)
```

El último método en este ejemplo es una propiedad .

La referencia de la instancia del modelo tiene una lista completa de métodos asignados automáticamente a cada modelo. Puede anular la mayoría de estos (consulte la sobreescritura de los métodos de modelo predefinidos, a continuación), pero hay un par que casi siempre querrá definir:

__str__()

Un "método mágico" de Python que devuelve una representación de cadena de cualquier objeto. Esto es lo que Python y Django usarán cada vez que una instancia de modelo deba forzarse y mostrarse como una cadena simple. En particular, esto sucede cuando muestra un objeto en una consola interactiva o en el administrador.

Siempre querrá definir este método; el valor predeterminado no es muy útil en absoluto.

get_absolute_url()

Esto le dice a Django cómo calcular la URL de un objeto. Django usa esto en su interfaz de administración, y cada vez que necesita encontrar una URL para un objeto.

Cualquier objeto que tenga una URL que lo identifique de forma única debe definir este método.

Sobreescribiendo métodos de modelo predefinidos

Hay otro conjunto de métodos de modelo que encapsulan un montón de comportamiento de la base de datos que querrá personalizar. En particular, a menudo querrá cambiar la forma de trabajo de **save()** y **delete()**.

Puede sobreescribir estos métodos (y cualquier otro método de modelo) para modificar el comportamiento.

Un caso de uso clásico para sobreescribir los métodos incorporados es si desea que suceda algo cada vez que guarda un objeto. Por ejemplo (consulte **save()** en la documentación de los parámetros que acepta):

```
from django.db import models

class Blog(models.Model):
  name = models.CharField(max_length=100)
  tagline = models.TextField()

def save(self, *args, **kwargs):
  do_something()
  super().save(*args, **kwargs)  # Call the "real" save() method.
```

```
        do_something_else()
```

También puede evitar guardar:

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super().save(*args, **kwargs)  # Call the "real" save() method.
```

Es importante recordar llamar al método de superclase **super().save(*args, **kwargs)**, ese es el negocio, para garantizar que el objeto aún se guarde en la base de datos. Si olvida llamar al método de la superclase, no se producirá el comportamiento predeterminado y no se tocará la base de datos.

También es importante que pase los argumentos que se pueden pasar al método del modelo; eso es lo que hace el ***args, **kwargs**. Django, de vez en cuando, ampliará las capacidades de los métodos de modelo incorporados, agregando nuevos argumentos. Si usa ***args, **kwargs** en las definiciones de su método, tiene la garantía de que su código admitirá automáticamente esos argumentos cuando se agreguen

Para más información por favor visite la documentación oficial que habla sobre el tema en https://docs.djangoproject.com/en/3.2/ref/models/instances/#model-instance-methods

Introducción a las vistas basadas en clases

Las vistas basadas en clases proporcionan una forma alternativa de implementar vistas como objetos de Python en lugar de funciones. No reemplazan las vistas basadas en funciones, pero tienen ciertas diferencias y ventajas en comparación con las vistas basadas en funciones:

- La organización del código relacionado con métodos HTTP específicos (**GET**, **POST**, etc.) puede abordarse mediante métodos separados en lugar de bifurcaciones condicionales.
- Las técnicas orientadas a objetos, como mixins (herencia múltiple), se pueden usar para factorizar elcódigo en componentes reutilizables.

La relación y el historial de vistas genéricas, vistas basadas en clases y vistas genéricas basadas en clases

Al principio solo existía el contrato de función de vista, Django pasó su función **HttpRequest** y esperaba un **HttpResponse**. Este fue el alcance de lo que proporcionó Django.

Al principio se reconoció que había modismos y patrones comunes en el desarrollo de la vista. Se introdujeron vistas genéricas basadas en funciones para abstraer estos patrones y facilitar el desarrollo de vistas para los casos comunes.

El problema con las vistas genéricas basadas en funciones es que, si bien cubrían bien los casos simples, no había manera de extenderlas o personalizarlas más allá de algunas opciones de configuración, lo que limitaba su utilidad en muchas aplicaciones del mundo real.

Las vistas genéricas basadas en clases se crearon con el mismo objetivo que las vistas genéricas basadas en funciones, para facilitar el desarrollo de vistas. Sin embargo, la forma en que se implementa la solución, mediante el uso de mixins, proporciona un conjunto de herramientas que da como resultado que las vistas genéricas basadas en clases sean más extensibles y flexibles que sus contrapartes basadas en funciones.

Si probó vistas genéricas basadas en funciones en el pasado y descubrió que faltaban, no debe pensar en las vistas genéricas basadas en clases como un equivalente basado en clases, sino como un nuevo enfoque para resolver los problemas originales que las vistas genéricas estaban destinadas a resolver.

El conjunto de herramientas de clases base y mixins que usa Django para crear

vistas genéricas basadas en clases está diseñado para brindar la máxima flexibilidad y, como tal, tiene muchos ganchos en forma de implementaciones de métodos predeterminados y atributos que es poco probable que le interesen en el uso más simple. casos. Por ejemplo, en lugar de limitarlo a un atributo basado en una clase para **form_class**, la implementación usa un método **get_form** que llama a un método **get_form_class** que, en su implementación predeterminada, devuelve el atributo **form_class** de la clase. Esto le brinda varias opciones para especificar qué formulario usar, desde un atributo hasta un gancho invocable completamente dinámico. Estas opciones parecen agregar complejidad hueca para situaciones simples, pero sin ellas, los diseños más avanzados estarían limitados.

Usando vistas basadas en clases

En esencia, una vista basada en clases le permite responder a diferentes métodos de solicitud HTTP con diferentes métodos de instancia de clase, en lugar de con un código de bifurcación condicional dentro de una función de vista única.

Entonces, donde el código para manejar HTTP **GET** en una función de vista se vería así:

```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponse('result')
```

En una vista basada en clases, esto se convertiría en:

```
from django.http import HttpResponsefrom django.views import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

Debido a que el solucionador de URL de Django espera enviar la solicitud y los argumentos asociados a una función invocable, no a una clase, las vistas basadas en clases tienen un método **as_view()** de clase que devuelve una función a la que se puede llamar cuando llega una solicitud de una URL que coincide con el patrón asociado. La función crea una instancia de la clase, llama **setup()** para inicializar sus atributos y luego llama a su método **dispatch(**. **Dispatch** mira la solicitud para determinar si es un **GET**, **POST** etc., y transmite la solicitud a un método coincidente si hay uno definido, o genera **HttpResponseNotAllowed** si no:

```python
# urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [
    path('about/', MyView.as_view()),
]
```

Vale la pena señalar que lo que devuelve su método es idéntico a lo que devuelve desde una vista basada en funciones, es decir, alguna forma de **HttpResponse**. Esto significa que los accesos directos u objetos http **TemplateResponse** son válidos para usar dentro de una vista basada en clases.

Si bien una vista mínima basada en clases no requiere ningún atributo de clase para realizar su trabajo, los atributos de clase son útiles en muchos diseños basados en clases y hay dos formas de configurar o establecer atributos de clase.

La primera es la forma estándar de Python de subclasificar y anular atributos y métodos en la subclase. Entonces, si su clase principal tuviera un atributo **greeting** como este:

```python
from django.http import HttpResponse
from django.views import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

Puede anular eso en una subclase:

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Otra opción es configurar atributos de clase como argumentos de palabras clave para la llamada **as_view()**en la URLconf:

```
urlpatterns = [
    path('about/', GreetingView.as_view(greeting="G'day")),
]
```

Usando mezclas (Mixins)

Los mixins son una forma de herencia múltiple en la que se pueden combinar comportamientos y atributos devarias clases principales.

Por ejemplo, en las vistas genéricas basadas en clases hay un mixin llamado **TemplateResponseMixin** cuyo propósito principal es definir el método **render_to_response()**. Cuando se combina con el comportamiento de la clase base **View**, el resultado es una clase **TemplateView** que enviará solicitudes a los métodos coincidentes apropiados (un comportamiento definido en la clase **View** base) y que tiene un método **render_to_response()** que usa un atributo **template_name** para devolver un objeto **TemplateResponse** (un comportamiento definido en el **TemplateResponseMixin**).

Los mixins son una forma excelente de reutilizar el código en varias clases, pero tienen un costo. Cuanto más disperso esté su código entre los mixins, más difícil será leer una clase secundaria y saber qué está haciendo exactamente, y más difícil será saber qué métodos de qué mixins sobreescribir si está subclasificando algo que tiene un árbol de herencia profundo.

Tenga en cuenta también que solo puede heredar de una vista genérica, es decir, solo una clase principal puede heredar **View** y el resto (si corresponde) debe ser mixins. Intentar heredar de más de una clase que hereda de **View**, por ejemplo, intentar usar un formulario en la parte superior de una lista y combinar **ProcessFormView** y **ListView** no funcionará como se esperaba.

Manejo de formularios con vistas basadas en clases

Una vista básica basada en funciones que maneja formularios puede verse así:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import MyForm

def myview(request):
    if request.method == "POST":
    form = MyForm(request.POST)
    if form.is_valid():
        # <process form cleaned data>
        return HttpResponseRedirect('/success/')
    else:
        form = MyForm(initial={'key': 'value'})
    return render(request, 'form_template.html', {'form': form})
```

Una vista similar basada en clases podría verse así:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views import View
from .forms import MyForm

class MyFormView(View):
    form_class = MyForm initial = {'key': 'value'}
    template_name = 'form_template.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})
    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
        # <process form cleaned data>
        return HttpResponseRedirect('/success/')
    return render(request, self.template_name, {'form': form})
```

Este es un caso mínimo, pero puede ver que luego tendría la opción de personalizar esta vista sobreescribiendo cualquiera de los atributos de clase, por ejemplo **form_class**, a través de la configuración de URLconf, o subclasificando y sobreescribiendo uno o más de los métodos (¡o ambos!).

Más sobre vistas basadas en clases

Una vista es un invocable que toma una solicitud y devuelve una respuesta. Esto puede ser más que una simple función, y Django proporciona un ejemplo de algunas clases que se pueden usar como vistas. Estos le permiten estructurar sus vistas y reutilizar el código aprovechando la herencia y los mixins. También hay algunas vistas genéricas para tareas que veremos más adelante, pero es posible que desee diseñar su propia estructura de vistas reutilizables que se adapte a su caso de uso. Para obtener detalles completos, consulte la documentación oficial de las vistas basadas en clases donde verá.

- Introducción a las vistas basadas en clases: https://docs.djangoproject.com/en/3.2/topics/class-based-views/intro/
- Vistas genéricas basadas en clases incorporadas: https://docs.djangoproject.com/en/3.2/topics/class-based-views/generic-display/
- Manejo de formularios con vistas basadas en clases: https://docs.djangoproject.com/en/3.2/topics/class-based-views/generic-editing/
- Uso de mixins con vistas basadas en clases: https://docs.djangoproject.com/en/3.2/topics/class-based-views/mixins/

Ejemplos básicos

Django proporciona clases de vista base que se adaptarán a una amplia gama de aplicaciones. Todas las vistas se heredan de la clase **View**, que maneja la vinculación de la vista a las URL, el envío del método HTTP y otras características comunes. **RedirectView** proporciona una redirección HTTP y **TemplateView** amplía la clase basepara que también represente una plantilla.

Uso en su URLconf

La forma más directa de usar vistas genéricas es crearlas directamente en su URLconf. Si solo está cambiando algunos atributos en una vista basada en clases, puede pasarlos a la llamada al método **as_view()** en sí:

```
from django.urls import path
from django.views.generic import TemplateView urlpatterns = [
    path('about/', TemplateView.as_view(template_name="about.html")),
]
```

Cualquier argumento pasado **as_view()** sobreescribirá los atributos establecidos en la clase. En este ejemplo, configuramos el **template_name** en la **TemplateView**. Se puede usar un patrón de reemplazo similar para el atributo **url** en **RedirectView**.

Subclasificación de vistas genéricas

La segunda forma, más poderosa, de usar vistas genéricas es heredar de una vista existente y sobreescribir atributos (como **template_name**) o métodos (como **get_context_data**) en su subclase para proporcionar nuevos valores o métodos. Considere, por ejemplo, una vista que solo muestra una plantilla, **about.html**. Django tiene una vista genérica para hacer esto, **TemplateView** por lo que podemos subclasificarlo y anular elnombre de la plantilla:

```
# some_app/views.py
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

Luego, debemos agregar esta nueva vista a nuestra URLconf. **TemplateView** es una clase, no una función, por lo que apuntamos la URL al método **as_view()** de clase, que proporciona una entrada similar a una función para las vistas basadas en clases:

```
# urls.py
from django.urls import path
from some_app.views import AboutView
urlpatterns = [
    path('about/', AboutView.as_view()),
]
```

Para obtener más información sobre cómo usar las vistas genéricas integradas, consulte el siguiente tema sobre vistas genéricas basadas en clases en la documentación oficial: https://docs.djangoproject.com/en/3.2/topics/class-based-views/generic-display/.