TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- Formularios en Django.
- La FormClass de Django.
- Procesamiento de formularios.
- Construyendo un formulario.
- Agregando la vista del formulario.
- Formularios y solicitud de protección contra la falsificación.
- Plantillas de formulario reutilizables.
- InlineFormsets y vinculación por llaves foráneas.
- Mensaje de errores.
- Models y Binding.

FORMULARIOS EN DJANGO

En HTML, un formulario es una colección de elementos dentro de `<form>...</form>` que permiten a un visitante hacer cosas como: ingresar texto, seleccionar opciones, manipular objetos o controles, entre otros, y luego enviar esa información de regreso al servidor.

Algunos de estos elementos de la interfaz de formulario (entrada de texto, o casillas de verificación) están integrados en el propio HTML. Otros son mucho más complejos; una interfaz que muestra un selector de fecha o le permite mover un control deslizante, o manipular controles que normalmente usará JavaScript y CSS, así como elementos HTML `<input>` para lograr estos efectos.

Además de sus elementos `<input>`, un formulario debe especificar dos cosas:

- Dónde: la URL a la que deben devolverse los datos correspondientes a la entrada del usuario.
- Cómo: el método HTTP debe devolver los datos.

Como ejemplo, el formulario de inicio de sesión para el administrador de Django contiene varios elementos `<input>: uno de type="text"` para el nombre de usuario; uno de `type="password"` para

la contraseña; y uno de `type="submit"` para Botón "Iniciar sesión". También contiene algunos campos de texto ocultos que el usuario no ve, y que Django usa para determinar qué hacer.

También le indica al navegador que los datos del formulario deben ser enviadas a la dirección URL especificada en el `<form> action atributo - /admin/ -`, y que debe ser enviada utilizando el mecanismo HTTP especificado por el método atributo - posterior.

Cuando se activa el elemento `<input type="submit" value="Log in">`, los datos se devuelven a `/admin/`. GET y POST son los únicos métodos HTTP para usar cuando se trata de formularios. El formulario de inicio de sesión de Django se devuelve utilizando el método POST, en el que el navegador agrupa los datos del formulario, los codifica para su transmisión, los envía al servidor, y luego recibe su respuesta.

GET, por el contrario, agrupa los datos enviados en una cadena y los utiliza para componer una URL. La URL contiene la dirección donde se deben enviar los datos, así como las claves y valores de datos. Puede ver esto en acción si realiza una búsqueda en la documentación de Django, que producirá una URL del formulario: https://docs.djangoproject.com/search/?q=forms&release=4.

GET y POST se usan típicamente para diferentes propósitos:

- Cualquier solicitud que pueda usarse para cambiar el estado del sistema; por ejemplo: una solicitud que realice cambios en la base de datos debe usar POST. GET debe usarse solo para solicitudes que no afectan el estado del sistema.
- GET tampoco sería adecuado para un formulario de contraseña, porque la contraseña aparecería en la URL y, por lo tanto, también en el historial del navegador y en los registros del servidor, todo en texto sin formato.
- Tampoco sería adecuado para grandes cantidades de datos, ni para datos binarios, como una imagen.
- Una aplicación web que usa solicitudes GET para formularios de administración es un riesgo de seguridad: puede ser fácil para un atacante imitar la solicitud de un formulario para obtener acceso a partes sensibles del sistema.
- POST, junto con otras protecciones como la protección CSRF de Django, ofrece más control sobre el acceso.

- GET es adecuado para cosas como un formulario de búsqueda web, porque las URL que representan una solicitud GET se pueden marcar, compartir, o volver a enviar fácilmente.

Manejar formularios es algo complejo. Considere el administrador de Django, donde se deben preparar numerosos elementos de datos de varios tipos diferentes, para mostrarlos en un formulario, renderizados como HTML, editados usando una interfaz conveniente, devueltos al servidor, validados y limpios, y luego guardados o transmitidos para su posterior procesamiento.

La funcionalidad de formulario de Django puede simplificar y automatizar vastas porciones de este trabajo, y también hacerlo de manera más segura de lo que la mayoría de los programadores podrían hacer en el código que escribieron ellos mismos.

Django se ocupa de tres partes distintas del trabajo relacionado con las formas:

- Preparando y reestructurando los datos para que estén listos para su presentación.
- Creando formularios HTML para los datos.
- Recibir y procesar los formularios y datos presentados por el cliente.

Es posible escribir código que haga todo esto manualmente, pero Django puede encargarse de hacerlo automáticamente.

FORMS DE DJANGO V/S FORMULARIOS HTML

Los formularios de Django son una construcción del lado del servidor. Los formularios HTML son una construcción del lado del cliente. Son cosas diferentes, a pesar de que la primera renderiza la segunda si quieres hacerlo.

Con respecto a cómo diseñar los formularios HTML Django generados, puede hacerlo usted mismo, agregando clases a los widgets y/o CSS semántico a su interfaz. Si se sigue este camino, se deben crear los propios widgets personalizados, subclasificando los predeterminados. Luego, puede agregar clases y otros atributos HTML a la interfaz generada en un solo lugar.

Si desea aprovechar los marcos frontend existentes, se recomienda usar Crispy Forms. Puede representar y diseñar automáticamente sus formularios utilizando marcos como Bootstrap, o Material Design. Simplemente seleccione el paquete de plantillas que desea usar.

ENLACE DE LOS FORMULARIOS CON LA VISTA

El manejo de formularios de Django es igual que cualquier otro programa con la estructura MVC: la vista recibe una solicitud, realiza las acciones requeridas, incluida la lectura de datos de los modelos, luego genera y devuelve una página HTML. Lo que complica más las cosas es que el servidor también debe poder procesar los datos proporcionados por el usuario y volver a mostrar la página si hay algún error.

Esta relación entre los formularios y la vista consiste en una serie de sucesos, o pasos, que el programa debe seguir, por ejemplo:

- Mostrar el formulario predeterminado la primera vez que lo solicite el usuario.
- El formulario puede contener campos en blanco si está creando un nuevo registro, o se puede completar previamente con valores iniciales.
- Recibe datos de una solicitud de envío y los vincula al formulario.
- Limpia y valida si los valores ingresados son apropiados para el campo (por ejemplo, que estén en el intervalo de fechas correcto, que no sean demasiado cortos ni demasiado largos, etc.)
- Si algún dato no es válido, vuelve a mostrar el formulario, esta vez con los valores completados por el usuario y los mensajes de error para los campos problemáticos.
- Si todos los datos son válidos, realice las acciones necesarias (como guardar los datos, enviar un correo electrónico, devolver el resultado de una búsqueda, cargar un archivo, etc.).
- Una vez completadas todas las acciones, redirige el usuario a otra página.

LA FORMCLASS DE DJANGO

En el corazón de este sistema de componentes se encuentra la clase Form de Django. De la misma manera que un modelo de Django describe la estructura lógica de un objeto, su comportamiento y la forma en que nos representan sus partes, una clase Form describe un formulario y determina cómo funciona y cómo aparece.

De forma similar a que los campos de una clase de modelo se asignan a los campos de la base de datos, los campos de una clase de formulario se asignan a `<input> elementos <input>` de formulario

HTML. (Un `ModelForm` asigna los campos de una clase de modelo a elementos `<input>` de formulario HTML a través de un Formulario; esto es en lo que se basa el administrador de Django).

Los campos de un formulario son en sí mismos, clases; administran los datos del formulario, y realizan la validación cuando éste se envía. Un `DateField` y un `FileField` manejan tipos de datos muy diferentes, y tienen que hacer cosas diferentes con ellos.

Un campo de formulario se representa para un usuario en el navegador como un "`widget`" HTML, una pieza de maquinaria de interfaz de usuario. Cada tipo de campo tiene una clase de widget predeterminada adecuada, pero se puede anular según sea necesario.

PROCESAMIENTO DE FORMULARIOS

Cuando se representa un objeto en Django, generalmente:

- Obtenerlo en la vista (buscarlo en la base de datos, por ejemplo)
- Pasarla al contexto de la plantilla.
- Expandirlo a un marcado HTML, usando variables de plantilla.

Renderizar un formulario en una plantilla implica casi el mismo trabajo que renderizar cualquier otro tipo de objeto, pero hay algunas diferencias clave.

En el caso de una instancia de modelo que no contenía datos, rara vez sería útil hacer algo con ella en una plantilla. Por otro lado, tiene mucho sentido representar un formulario vacío, eso es lo que hacemos cuando queremos que el usuario lo complete.

Entonces, cuando manejamos una instancia de modelo en una vista, generalmente la recuperamos de la base de datos. Cuando tratamos con un formulario, lo instanciamos en la vista. Cuando instanciamos una forma, podemos optar por dejarla vacía o pre-cargada. Por ejemplo, con:

- Datos de una instancia de modelo guardada (como en el caso de los formularios de administración para la edición)
- Datos que hemos cotejado de otras fuentes.
- Datos recibidos de un formulario HTML anterior.

El último de estos casos es el más interesante, porque es lo que hace posible que los usuarios no solo lean un sitio web, sino que también le envíen información.

CONSTRUYENDO UN FORMULARIO

Suponga que desea crear un formulario simple en su sitio web para obtener el nombre del usuario. Necesitarías algo como esto en tu plantilla:

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{
current_name }}">
    <input type="submit" value="OK">
</form>
```

Esto le indica al navegador que devuelva los datos del formulario a la URL `/your-name/`, utilizando el método POST. Mostrará un campo de texto, etiquetado "Su nombre:", y un botón marcado "OK". Si el contexto de la plantilla contiene una variable `current_name`, se usará para completar el campo `your_name`.

Necesitará una vista que represente la plantilla que contiene el formulario HTML, y que puede proporcionar el campo `current_name`, según corresponda.

Cuando se envía el formulario, la solicitud POST que se envía al servidor contendrá los datos del formulario. Ahora también necesitará una vista correspondiente a esa `/your-name/` URL, que encontrará los pares clave/valor apropiados en la solicitud y luego los procesará.

Esta es una forma muy simple. En la práctica, un formulario puede contener docenas o centenares de campos, muchos de los cuales pueden necesitar ser rellenados previamente, y podemos esperar que el usuario trabaje a través del ciclo de edición y envío varias veces antes de concluir la operación.

Podemos requerir de alguna validación para que ocurra en el navegador, incluso antes de que se envíe el formulario, podemos querer usar campos mucho más complejos, que permitan al usuario hacer cosas como elegir fechas de un calendario y así sucesivamente. En este punto, es mucho más fácil hacer que Django haga la mayor parte de este trabajo por nosotros.

LA CLASE DE FORM

Ya sabemos cómo queremos que se vea nuestro formulario HTML. Nuestro punto de partida en Django es el siguiente:

```
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```

Esto define una clase de `Form` con un solo campo (`your_name`). Hemos aplicado una etiqueta amigable para el hombre al campo, que aparecerá en la `<label>` cuando se procese (aunque en este caso, la label que especificamos es en realidad la misma que se generaría automáticamente si la hubiéramos omitido).

La longitud máxima permitida del campo se define por `max_length`. Esto hace dos cosas. Pone un `maxlength="100"` en el HTML `<input>` (por lo que el navegador debe evitar que el usuario ingrese más que ese número de caracteres en primer lugar). También significa que cuando Django recibe el formulario desde el navegador, validará la longitud de los datos.

Una instancia de `Form` tiene un método `is_valid()`, que ejecuta rutinas de validación para todos sus campos. Cuando se llama a este método, si todos los campos contienen datos válidos, hará lo siguiente:

- Devolver un `True`.
- Coloque los datos del formulario en su atributo `cleaned_data`.

Toda la forma, cuando se representa por primera vez, se verá como:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100"
required>
```

Tenga en cuenta que no incluye las etiquetas `<form>`, o un botón de enviar. Tendremos que proporcionarlos nosotros mismos en la plantilla.

AGREGANDO LA VISTA DEL FORMULARIO

Los datos del formulario que se envían a un sitio web de Django se procesan mediante una vista, generalmente la misma vista que publicó el formulario. Esto nos permite reutilizar parte de la misma lógica. Para manejar la forma, necesitamos instanciarla en la vista para la URL donde queremos que se publique:

VIEWS.PY

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

from .forms import NameForm

def get_name(request):
# si se trata de una solicitud POST, necesitamos procesar los datos del formulario
    if request.method == 'POST':
        # crear una instancia de formulario y completarla con los datos de la solicitud:
        form = NameForm(request.POST)
        # comprobar si es válido:
        if form.is_valid():
            # procesar los datos en form.cleaned_data según sea necesario
            # ...
            # redirigir a una nueva URL:
            return HttpResponseRedirect('/thanks/')

    # si un GET (o cualquier otro método) crearemos un formulario en blanco
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})
```

Si llegamos a esta vista con una solicitud GET, creará una instancia de formulario vacía y la colocará en el contexto de la plantilla que se representará. Esto es lo que podemos esperar que suceda la primera vez que visitemos la URL. Si el formulario se envía mediante una solicitud POST, la vista volverá a crear una instancia de formulario, y la completará con los datos de la solicitud: `form = NameForm(request.POST)` Esto se denomina "datos vinculantes para el formulario" (ahora es una forma encuadernada).

Llamamos al método `is_valid()` del formulario; Si no es True, volvemos a la plantilla con el formulario. Esta vez, el formulario ya no está vacío (sin vincular), por lo que el formulario HTML se completará con los datos enviados previamente, donde se puede editar y corregir según sea necesario.

Si `is_valid()` es True, ahora podremos encontrar todos los datos de formulario validados en su atributo `cleaned_data`. Podemos usar estos datos para actualizar la base de datos, o hacer otro procesamiento antes de enviar un redireccionamiento HTTP al navegador indicándole dónde ir a continuación.

AGREGANDO EL TEMPLATE DE DESPLIEGUE

No necesitamos hacer mucho en nuestra plantilla `name.html`:

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

Todos los campos del formulario, y sus atributos, se descomprimirán en el marcado HTML a partir de ese `{{form}}` por el lenguaje de plantilla de Django.

PLANTILLAS DE FORMULARIOS

Todo lo que necesitas hacer para convertir tu formulario en una plantilla es colocar la instancia del formulario en el contexto de la plantilla. Entonces, si tu formulario se llama “form” en el contexto, `{{form}}` representará sus elementos `<label>` y `<input>` apropiadamente. No olvides que la salida de un formulario no incluye las etiquetas `<form>` circundantes ni el control de envío del formulario. Tendrás que proporcionarlos tú mismo.

FORMULARIOS Y SOLICITUD DE PROTECCIÓN CONTRA LA FALSIFICACIÓN

En Django se envía con una protección fácil de usar contra falsificaciones de solicitudes entre sitios. Al enviar un formulario a través de POST con la protección CSRF habilitada, debe usar la etiqueta de plantilla `csrf_token`, como en el ejemplo anterior. Sin embargo, dado que la protección CSRF no está directamente vinculada a los formularios en las plantillas.

TIPOS DE ENTRADA DE HTML5 Y VALIDACIÓN DEL NAVEGADOR

Si su formulario incluye un `URLField`, un `EmailField`, o cualquier tipo de campo entero, Django usará los tipos de entrada de URL, email y number HTML5. Por defecto, los navegadores pueden aplicar su propia validación en estos campos, que pueden ser más estrictos que la validación de Django. Si desea deshabilitar este comportamiento, establezca el atributo `novalidate` en la etiqueta del formulario, o especifique un widget diferente en el campo, como `TextInput`.

RENDERIZAR LOS CAMPOS DE FORMA MANUAL

No tenemos que dejar que Django presente los campos del formulario; podemos hacerlo manualmente si lo deseamos (lo que nos permite reordenar los campos, por ejemplo). Cada campo está disponible como un atributo del formulario utilizando `{{form.name_of_field}}`, y en una plantilla de Django, se representará de manera adecuada. Por ejemplo:

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email
address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

Los elementos completos de `<label>` también se pueden generar utilizando `label_tag ()`. Por ejemplo:

```
<div class="fieldWrapper">
```

```
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

PLANTILLAS DE FORMULARIO REUTILIZABLES

Si su sitio usa la misma lógica de representación para los formularios en varios lugares, puede reducir la duplicación guardando el bucle del formulario en una plantilla independiente, y anulando el atributo `template_name` de los formularios para representar el formulario con la plantilla personalizada. El siguiente ejemplo dará como resultado que `{{ form }}` se represente como el resultado de la plantilla `form_snippet.html`.

En sus plantillas:

```
# En tu plantilla:
{{ form }}

# En form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

En su formulario:

```
class MyForm(forms.Form):
    template_name = 'form_snippet.html'
    ...
```

INLINEFORMSETS Y VINCULACIÓN POR LLAVES FORÁNEAS

Los conjuntos de formas en línea son una pequeña capa de abstracción en la parte superior de los conjuntos de formas modelo. Estos simplifican el caso de trabajar con objetos relacionados a través de una clave foránea (`foreign key`). Supongamos que tiene estos dos modelos:

```
from django.db import models

```

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
```

Si quiere crear un formulario que permita editar libros que pertenezcan a un autor en particular, puede hacerlo:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('title',))
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)
```

`BookFormSet` prefijo es `'book_set'` (`<model name>_set`). Si Book es `ForeignKey` de Author, tiene una `related_name`, que se utiliza en su lugar.

Nota: inlineformset_factory() usa modelformset_factory(), y marca can_delete=True.

MENSAJES DE ERRORES

Hasta ahora no hemos tenido que preocuparnos por cómo mostrar los errores de formulario, porque eso es lo que nos ocupamos. En este ejemplo, hemos tenido que asegurarnos de que nos ocupamos de los errores de cada campo, y también de los del formulario en su conjunto. Tenga en cuenta `{{form.non_field_errors}}` en la parte superior del formulario, y la plantilla de búsqueda de errores en cada campo. El uso de `{{form.name_of_field.errors}}` muestra una lista de errores de formulario, representada como una lista desordenada. Esto podría verse así:

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

La lista tiene una clase CSS de lista de errores para permitirle diseñar su apariencia. Si desea personalizar aún más la visualización de errores, puede hacerlo mediante un bucle sobre ellos:

```
{% if form.subject.errors %}
    <ol>
```

```
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

Los errores que no son de campo (y / o los errores de campo oculto que se muestran en la parte superior del formulario cuando se usan ayudantes como `form.as_p()`) se mostrarán con una clase adicional de `nonfield`, para ayudar a distinguirlos de los errores específicos del campo. Por ejemplo, `{{form.non_field_errors}}` se vería así:

```
<ul class="errorlist nonfield">
    <li>Generic validation error.</li>
</ul>
```

MODELS Y BINDING

Si está creando una aplicación basada en bases de datos, es probable que tenga formularios que se correspondan estrechamente con los modelos de Django. Por ejemplo: es posible que tenga un modelo de `BlogComment`, y desee crear un formulario que permita a las personas enviar comentarios. En este caso, sería redundante definir los tipos de campo en su formulario, porque ya ha definido los campos en su modelo.

Por esta razón, Django proporciona una clase auxiliar que le permite crear una clase `Form`, a partir de un modelo Django. Por ejemplo:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

La clase de `Form` generada tendrá un campo de formulario para cada campo de modelo especificado, en el orden especificado en el atributo de `fields`. Cada campo modelo tiene un campo de formulario predeterminado correspondiente. Por ejemplo, un `CharField` en un modelo se representa como un `CharField` en un formulario. Un modelo `ManyToManyField` se representa como `MultipleChoiceField`.

Además, cada campo de forma generado tiene atributos establecidos de la siguiente manera:

Si el campo modelo tiene en `blank=True`, entonces `required` se establece en False en el campo del formulario. De lo contrario, `required=True`. La `label` del campo de formulario se establece en el nombre `verbose_name` del campo modelo, con el primer carácter en mayúscula. El texto de `help_text` del campo de formulario se establece en el `help_text` de ayuda del campo modelo.

Si el campo modelo tiene `choices` establecidas, entonces el widget del campo de formulario se establecerá en `Select`, con opciones provenientes de las `choices` del campo modelo. Las opciones normalmente incluirán la opción en blanco que se selecciona por defecto. Si el campo es obligatorio, esto obliga al usuario a realizar una selección. La opción en blanco no se incluirá si el campo del modelo tiene `blank=False`, y un valor default explícito (el cual se seleccionará inicialmente en su lugar).

Por ejemplo, considere este conjunto de modelos:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = [
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
]
class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name

class Book(models.Model):
```

```
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['name', 'authors']
```

Con estos modelos, las subclases `ModelForm` anteriores serían más o menos equivalentes a esto (la única diferencia es el método `save()`, que discutiremos en un momento):

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = [
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
]

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['name', 'authors']
```

Cada `ModelForm` también tiene un método `save()`. Este método crea y guarda un objeto de base de datos a partir de los datos vinculados al formulario. Una subclase de `ModelForm` puede aceptar una instancia de modelo existente como `instance` argumento de palabra clave; si se proporciona esto, `save()` actualizará esa instancia. Si no se proporciona, `save()` creará una nueva instancia del modelo especificado:

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Cree una instancia de formulario a partir de datos POST.
>>> f = ArticleForm(request.POST)

# Guarde un nuevo objeto Artículo a partir de los datos del formulario.
>>> new_article = f.save()

# Crea un formulario para editar un artículo existente, pero usa
# Datos POST para completar el formulario.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Tenga en cuenta que si el formulario no ha sido validado, llamando a `save()`, lo hará marcando `form.errors`. Emite un `ValueError` si los datos en el formulario no se validan, es decir, si `form.errors` se evalúa como `True`.