Contenidos a Trabajar

1. Widgets
   - Especificando Widgets
   - Argumentos para Widgets
   - Estilo de Instancias de widgets
   - Estilizar clases de widgets
2. Validaciones de formularios y campos
3. El Marco de Mensajes
   - Habilitación de Mensajes
   - Niveles de Mensajes
   - Usando mensajes de vistas y templates
   - Visualización de mensajes

Widgets

Un widget es la representación de Django de un elemento de entrada HTML. El widget maneja la representación del HTML y la extracción de datos de un diccionario GET/POST que corresponde al widget.

El HTML generado por los widgets integrados utiliza la sintaxis de HTML5, con orientación

<!DOCTYPE html>. Por ejemplo, utiliza atributos booleanos **checked** como en lugar del estilo XHTMLde **checkedchecked='checked'.**

Importante

Los widgets no deben confundirse con los campos de formulario . Los campos de formulario se ocupan de la lógica de la validación de entrada y se utilizan directamente en las plantillas. Los widgets se ocupan de la representación de elementos de entrada de formulario HTML en la página web y la extracción de datos enviados sin procesar. Sin embargo, los widgets deben asignarse a los campos de formulario.

Especificando Widgets

Cada vez que especifique un campo en un formulario, Django usará un widget predeterminado que sea apropiado para el tipo de datos que se mostrarán. Para encontrar qué widget se usa en qué campo, consulte la documentación sobre las clases de campo incorporadas en el sitio de Django:

https://docs.djangoproject.com/es/3.2/ref/forms/fields/

Sin embargo, si desea usar un widget diferente para un campo, puede usar el argumento widget en ladefinición del campo. Por ejemplo:

```
from django import forms
class CommentForm(forms.Form):
        name = forms.CharField()
        url = forms.URLField()
        comment = forms.CharField(widget=forms.Textarea)
```

Esto especificaría un formulario con un comentario que usa un widget **Textarea** más grande, en lugar delwidget predeterminado **TextInput**.

Argumentos para Widgets

Muchos widgets tienen argumentos adicionales opcionales; se pueden configurar al definir el widget en el campo. En el siguiente ejemplo, el atributo **years** se establece para un **SelectDateWidget**:

```
from django import forms

BIRTH_YEAR_CHOICES = ['1980', '1981',
'1982']FAVORITE_COLORS_CHOICES = [
    ('blue', 'Blue'),
    ('green', 'Green'),
    ('black', 'Black'),
]
class SimpleForm(forms.Form):
    birth_year                                                    =
    forms.DateField(widget=forms.SelectDateWidget(years=BIRTH_YEAR_CHOICES))
    favorite_colors =Forms.MultipleChoiceField(required=False,
        widget=forms.CheckboxSelectMultiple, choices=FAVORITE_COLORS_CHOICES,
    )
```

Para obtener más información sobre que widgets están disponibles y que argumentos aceptan, consulte el sitio oficial en la sección de widgets integrados: https://docs.djangoproject.com/es/3.2/ref/forms/widgets/

Estilo de Instancias de widgets

Si desea que una instancia de widget se vea diferente de otra, deberá especificar atributos adicionales en el momento en que se instancia el objeto de widget y se asigna a un campo de formulario (y quizás agregue algunas reglas a sus archivos CSS).

En una página web real, probablemente no desee que todos los widgets tengan el mismo aspecto. Es posible que desee un elemento de entrada más grande para el comentario y que el widget de "nombre" tenga alguna clase CSS especial. También es posible especificar el atributo "tipo" para aprovechar los nuevos tipos de entrada de HTML5. Para hacer esto, usa el argumento Widget.attrs al crear el widget:

```
class CommentForm(forms.Form):

name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'})) url =
forms.URLField()
comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

Estilizar clases de widgets

Con los widgets, es posible agregar activos (**css** y **javascript**) y personalizar más profundamente su apariencia ycomportamiento.

En pocas palabras, deberá subclasificar el widget y definir una clase interna «Media» o crear una propiedad «media».

Para más información sobre dicho manejo, revisa la documentación oficial y las distintas forma de aplicarlo:

https://docs.djangoproject.com/es/3.2/topics/forms/media/

Validaciones de formularios y campos

La validación del formulario ocurre cuando se limpian los datos. Si desea personalizar este proceso, hay varios lugares para realizar cambios, cada uno con un propósito diferente. Se ejecutan tres tipos de métodos de limpieza durante el procesamiento del formulario. Estos normalmente se ejecutan cuando llama al método **is_valid()** en un formulario. Hay otras cosas que también pueden activar la limpieza y la validación (acceder al atributo **errors** o llamar directamente **full_clean()**), pero normalmente no serán necesarias.

En general, cualquier método de limpieza puede generar **ValidationError** si hay un problema con los datos que está procesando, pasando la información relevante al constructor **ValidationError**. Vea a continuación las mejores prácticas para lanzar **ValidationError**. Si no se genera **ValidationError**, el método debe devolver los datos limpios (normalizados) como un objeto de Python.

La mayoría de las validaciones se pueden realizar mediante validadores, ayudantes que se pueden reutilizar. Los validadores son funciones (o invocables) que toman un solo argumento y generan

ValidationError en una entrada no válida. Los validadores se ejecutan después de que se hayan ejecutado paralos campos los métodos **to_python** y **validate.**

La validación de un formulario se divide en varios pasos, que se pueden personalizar o anular:

- El método **to_python()** en un **Field** es el primer paso en cada validación. Coacciona el valor a un tipo de datos correcto y aumenta lanza un **ValidationError** si eso no es posible. Este método acepta el valor bruto del widget y devuelve el valor convertido. Por ejemplo, un **FloatField** convertirá los datosen un **float** de Python o lanzará **ValidationError**.
- El método **validate()** en un **Field** maneja la validación específica de campo que no es adecuada para un validador. Toma un valor que ha sido forzado a un tipo de datos correcto y lanza un **ValidationError** en cualquier error. Este método no devuelve nada y no debería alterar el valor. Debe sobrescribirlo para manejar la lógica de validación que no puede o no quiere poner en un validador.
- El método **run_validators()** en **Field** ejecuta todos los validadores del campo y agrega todos los errores en un solo **ValidationError**. No debería necesitar sobrescribir este método.

- El método **clean()** en una subclase **Field** es responsable de ejecutar **to_python()**, **validate()**

  y **run_validators()** en el orden correcto y propagar sus errores. Si, en algún momento, alguno de los métodos genera **ValidationError**, la validación se detiene y se genera ese error. Este método devuelve los datos limpios, que luego se insertan en el diccionario **cleaned_data** del formulario.

- El método **clean_<fieldname>()** se llama en una subclase de formulario, donde **<fieldname>** se reemplaza con el nombre del atributo de campo de formulario. Este método realiza cualquier limpieza que sea específica de ese atributo en particular, sin relación con el tipo de campo que es. A este método no se le pasa ningún parámetro. Deberá buscar el valor del campo en **self.cleaned_data** y recordar que será un objeto de Python en este punto, no la cadena original enviada en el formulario (estará **cleaned_data** porque el método de campo general **clean()**, arriba, ya ha limpiado el datos una vez).

  Por ejemplo, si desea validar que el contenido de una **CharField** llamado **serialnumber** es

  único, **clean_serialnumber()** sería el lugar adecuado para hacerlo. No necesita un campo específico (es un **CharField**), pero desea una pieza de validación específica del campo de formulario y, posiblemente, limpiar/normalizar los datos.

  El valor de retorno de este método reemplaza el valor existente en **cleaned_data**, por lo que debe ser el valor del campo **cleaned_data** (incluso si este método no lo cambió) o un nuevo valor limpio.

- El método **clean()** de la subclase de formulario puede realizar una validación que requiere acceso a múltiples campos de formulario. Aquí es donde puede marcar comprobaciones como «si se proporciona el campo **A**, el campo **B** debe contener una dirección de correo electrónico válida». Este método puede devolver un diccionario completamente diferente si lo desea, que se utilizará como archivo **cleaned_data**.

  Dado que los métodos de validación de campo se han ejecutado en el momento **clean()** en que se llama, también tiene acceso al atributo **errors** del formulario que contiene todos los errores generados por la limpieza de campos individuales.

  Tenga en cuenta que cualquier error generado por su sobrescritura de **Form.clean()** no se asociará con ningún campo en particular. Van a un

«campo» especial (llamado **____all____**), al que puede acceder a través del método **non_field_errors()** si lo necesita. Si desea adjuntar errores a un campo específicodel formulario, debe llamar al **add_error()**.

También tenga en cuenta que existen consideraciones especiales al anular el método **clean()**de una subclase **ModelForm**, pero se verá mas adelante en el curso.

Estos métodos se ejecutan en el orden indicado anteriormente, un campo a la vez. Es decir, para cada campo del formulario (en el orden en que se declaran en la definición del formulario), se ejecuta el método **Field.clean()** (o su sobrescritura), luego **clean_<fieldname>()**. Finalmente, una vez que esos dos métodos se ejecutan para cada campo, el método **Form.clean()**, o su sobrescritura, se ejecuta ya sea que losmétodos anteriores hayan generado errores o no.

Como se mencionó, cualquiera de estos métodos puede generar un **ValidationError**. Para cualquier campo, si el método **Field.clean()** genera un **ValidationError**, no se llama a ningún método de limpieza específico del campo. Sin embargo, los métodos de limpieza para todos los campos restantes aún se ejecutan.

Para ver más detalle sobre la generación de errores y ejemplos, visite la documentación oficial:https://docs.djangoproject.com/es/3.2/ref/forms/validation/

El Marco de Mensajes

Muy comúnmente en las aplicaciones web, debe mostrar un mensaje de notificación único (también conocido como "mensaje flash") al usuario después de procesar un formulario u otros tipos de entrada del usuario.

Para esto, Django brinda soporte completo para mensajes basados en cookies y sesiones, tanto para usuarios anónimos como autenticados. El marco de mensajes le permite almacenar mensajes temporalmente en una solicitud y recuperarlos para mostrarlos en una solicitud posterior (generalmente la siguiente). Cada mensaje tiene una etiqueta específica **level** que determina su prioridad (p. ej. **info**, **warning**, o **error**).

Habilitación de Mensajes

Los mensajes se implementan a través de una clase de middleware y el procesador decontexto correspondiente.

El valor predeterminado creado por **django-admin startproject** en **settings.py** ya contiene todas las configuraciones necesarias para habilitar la funcionalidad de mensajes:

- **'django.contrib.messages'**está en **INSTALLED_APPS**
- **MIDDLEWARE** contiene **'django.contrib.sessions.middleware.SessionMiddleware'** y 'django.contrib.messages.middleware.MessageMiddleware'.

  El backend de almacenamiento predeterminado se basa en sesiones. Por eso **SessionMiddleware** debe estar habilitado y aparecer antes de **MessageMiddleware** en **MIDDLEWARE**.
- La opción **'context_processors'** del **DjangoTemplates** backend definida en su configuración

  TEMPLATES contiene 'django.contrib.messages.context_processors.messages'.

Si no desea utilizar mensajes, puede eliminar **'django.contrib.messages'** de su **INSTALLED_APPS**, la línea **MessageMiddleware** de **MIDDLEWARE** y el procesador de contexto **messages** de **TEMPLATES**.

Niveles de Mensajes

El marco de mensajes se basa en una arquitectura de nivel configurable similar a la del módulo de registro de Python. Los niveles de mensajes le permiten agrupar mensajes por tipo para que puedan filtrarse o mostrarse de manera diferente en vistas y plantillas.

Los niveles integrados, desde los que se pueden importar desde **django.contrib.messages** directamente, son:

Constante	Objetivo
DEBUG	Mensajes relacionados con el desarrollo que se ignorarán (o eliminarán) en una implementación deproducción
INFO	Mensajes informativos para el usuario
SUCCESS	Una acción fue exitosa, por ejemplo, "Su perfil se actualizó con éxito"
WARNING	No ocurrió una falla pero puede ser inminente
ERROR	Una acción **no** tuvo éxito o se produjo algún otro error

La configuración **MESSAGE_LEVEL** se puede usar para cambiar el nivel mínimo registrado (o se puede cambiar por solicitud). Se ignorarán los intentos de agregar mensajes de un nivel inferior a este.

Usando mensajes en vistas y templates

add_message(***solicitud*** **,** ***nivel*** **,** ***mensaje*** **,** ***extra_tags*** **= '' ,** ***fail_silently*** **= False)**

Para agregar un mensaje, llame a lo siguiente:

```
from django.contrib import messages messages.add_message(request,
messages.INFO, 'Hello world.')
```

Algunos métodos abreviados proporcionan una forma estándar de agregar mensajes con etiquetas de uso común (que generalmente se representan como clases HTML para el mensaje):

```
messages.debug(request, '%s SQL statements were executed.' % count)
messages.info(request, 'Three credits remain in your account.')
```

```
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

Visualización de mensajes

get_messages(*solicitud*)

En la plantilla, use algo como:

```
{% if messages %}
<ul class="messages">
        {% for message in messages %}
        <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{
        message }}</li>
        {% endfor %}
</ul>
{% endif %}
```

Si está utilizando el procesador de contexto, su plantilla debe representarse con una **RequestContext**. De lo contrario, asegúrese de que **messages** esté disponible para el contexto de la plantilla.

Incluso si sabe que solo hay un mensaje, aún debe repetir la secuencia **messages** porque, de lo contrario, el almacenamiento de mensajes no se borrará para la siguiente solicitud.

El procesador de contexto también proporciona una variable **DEFAULT_MESSAGE_LEVELS** que es una asignación de los nombres de nivel de mensaje a su valor numérico:

```
{% if messages %}
<ul class="messages">
        {% for message in messages %}
        <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
                {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}Important:
                {% endif %}
                {{ message }}
        </li>
        {% endfor %}
</ul>
{% endif %}
```

Para más información sobre el manejo de mensajes, vaya a la documentación oficial enhttps://docs.djangoproject.com/en/3.2/ref/contrib/messages/