HINTS

SISTEMA DE ADMINISTRACIÓN DE DJANGO

El administrador cuenta con los objetos `Question` con una interfaz de administración. Para esto, abrimos el archivo `polls/admin.py`, y lo editamos para que tenga el siguiente contenido:

```
polls/admin.py
from django.contrib import admin
from .models import Question
admin.site.register(Question)
```

Aquí se mostrará una página de listado (`change list`) de preguntas. Ésta muestra todas las preguntas en la base de datos, y nos permite elegir una para modificarla. Se destaca:

- El form es generado automáticamente a partir del modelo `Question`.
- Los diferentes tipos de campo del modelo (`DateTimeField`, `CharField`) se corresponden con el widget HTML apropiado. Cada tipo de campo sabe cómo mostrarse en el admin de Django.
- Cada `DateTimeField` tiene atajos JavaScript. La fecha tienen un atajo para “Hoy” (Today) y un popup de calendario, y la hora un “Ahora” (Now) y un popup que lista las horas más comunes.

Para el cierre de la página nos da algunas opciones:

- Guardar (Save) – Guarda los cambios, y nos devuelve a la página listado para este tipo de objeto.
- Grabar y continuar editando (Save and continue editing) – Guarda los cambios, y recarga la página del admin de este objeto.
- Grabar y agregar otro (Save and add another) – Guarda los cambios, y carga un form nuevo, en blanco, que permite agregar un nuevo objeto de este tipo.
- Eliminar (Delete) – Muestra una página de confirmación de borrado.

Django hace un gran trabajo al crear un sitio de administración básico usando la información de los modelos registrados:

- Cada modelo tiene una lista de registros individuales, identificados por la cadena creada por el método `__str__()` del modelo, y enlazados a vistas/formularios de detalle para edición. Por

defecto, esta vista de lista tiene un menú de acción en la parte superior, la cual puedes usar para realizar operaciones de eliminación masiva de los registros.

- Los formularios de detalle de registro del modelo para edición y adición de registros contienen todos los campos del modelo, organizados verticalmente en su orden de declaración.

Posteriormente, puedes personalizar la interfaz para hacerla incluso más fácil de usar. Algunas de las cosas que puedes hacer son:

Vistas de lista:

- Añadir campos e información adicional desplegada para cada registro.
- Añadir filtros para seleccionar qué registros se listan, basados en fechas u otros tipos de valores (ej. estado de préstamo del libro).
- Añadir opciones adicionales al menú `Action` en las vistas de lista, y elegir en qué lugar del formulario se despliega este menú.

Vistas de detalle:

- Elegir qué campos desplegar (o excluir), junto con su orden, agrupamiento, si son editables, el tipo de control a usarse, orientación, entre otros.
- Añadir campos relacionados a un registro para permitir la edición en cadena (ej. proveer la capacidad de añadir y editar registros de libros mientras estás creando su registro de autor).

INSTALAR EL PROYECTO DE AUTENTICACIÓN PERSONALIZADO DJANGO

Cree un proyecto Django llamado `djangoauth` usando el siguiente comando:

```
1 django-admin startproject DjangoAuth
```

Ejecute el siguiente comando para crear una nueva aplicación Django llamada `authen` en su proyecto:

```
1 cd DjangoAuth
2 python manage.py startapp authen
```

Abra la `carga/aplicación.Py`. Podemos ver la clase `AuthenConfig` (subclase de `django.apps.AppConfig`), que representa nuestra aplicación Django y su configuración:

```
from django.apps import AppConfig
class AuthenConfig(AppConfig):
    name = 'authen'
```

Abra la **configuración.Py**, localice **INSTALLED_APPS**, y añada:

```
INSTALLED_APPS = [
    ...
    'authen.apps.AuthenConfig',
]
```

MÉTODOS EN INSTANCIAS DE CONTENTTYPE

Cada instancia de **ContentType** tiene métodos que le permiten pasar de una instancia de **ContentType** al modelo que representa, o recuperar objetos de ese modelo:

```
ContentType.get_object_for_this_type(**kwargs)
```

Toma un conjunto de argumentos de búsqueda válidos para el modelo que **ContentType** representa, y realiza a **get()** lookup en ese modelo, devolviendo el objeto correspondiente.

```
ContentType.model_class()
```

Devuelve la clase de modelo representada por esta instancia de **ContentType**. Por ejemplo, podríamos buscar **ContentType** para el modelo de **User**:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> user_type = ContentType.objects.get(app_label='auth', model='user')
>>> user_type
<ContentType: user>
```

Y luego usarlo para consultar un **User** particular, o para obtener acceso a la clase de modelo **User**:

```
>>> user_type.model_class()
<class 'django.contrib.auth.models.User'>
>>> user_type.get_object_for_this_type(username='Guido')
<User: Guido>
```

Juntos, **get_object_for_this_type()** y **model_class()** permiten dos casos de uso extremadamente importantes:

- Con estos métodos, puede escribir código genérico de alto nivel que realice consultas en cualquier modelo instalado; en lugar de importar y usar una sola clase de modelo específica, puede pasar una `app_label` y un `model`, o a una búsqueda de `ContentType` en tiempo de ejecución, y luego trabajar con el modelar clase o recuperar objetos de ella.
- Puede relacionar otro modelo con `ContentType` como una forma de vincular las instancias de éste a clases de modelos particulares, y utilizar los métodos para obtener acceso a esas clases de modelos.

Varias de las aplicaciones incluidas de Django hacen uso de esta última técnica. Por ejemplo, el sistema de permisos en el marco de autenticación de Django utiliza un modelo de `Permission` con una clave foránea para `ContentType`; esto permite que Permission represente conceptos como "puede agregar una entrada de blog", o "puede eliminar una noticia".

MANEJO DE SESIONES

Algunos aspectos interesantes de la gestión interna de las sesiones:

- El diccionario de la sesión acepta cualquier objeto Python capaz de ser serializado con `pickle`. (Módulo `pickle`, incluido en la librería estándar de Python).
- Los datos de la sesión se almacenan en una tabla en la base de datos, llamada `django_session`.
- Los datos de la sesión son suministrados bajo demanda. Si nunca accedes al atributo `request.session`, Django nunca accederá a la base de datos.
- Django sólo envía la cookie si tiene que hacerlo. Si no modificas ningún valor de la sesión, no reenvía la cookie (a no ser que se haya definido `SESSION_SAVE_EVERY_REQUEST` como `True`).
- El entorno de sesiones de Django se basa entera y exclusivamente en las cookies. No almacena la información de la sesión en las URL, como recurso extremo en el caso de que no se puedan utilizar las cookies, como hacen otros entornos (PHP, JSP).

MENSAJES PERSONALIZADOS

En algunos casos particulares, será necesario crear un mensaje personalizado. Recuerda que los niveles de cada mensaje son números enteros, así que puedes definir un valor constante que puedes agregar a tus mensajes.

```
CRITICAL = 50
def mi_vista(request):
    messages.add_message(request, CRITICAL, 'Error grave')
```

Para asignarle su respectiva etiqueta dirígete a `settings.py`, y agrega lo siguiente a tu `MESSAGE_TAGS`:

```
MESSAGE_TAGS = {
    ...
    50: 'alert-danger',
}
```

VISTA DEL DESARROLLO DE LOS ARCHIVOS ESTÁTICOS

Las herramientas de archivos estáticos están diseñadas principalmente para ayudar a que los archivos estáticos se implementen con éxito en la producción. Esto generalmente significa un servidor de archivos estático separado y dedicado, que es una gran sobrecarga para meterse cuando se desarrolla localmente. Por lo tanto, la aplicación staticfiles se envía con una vista auxiliar rápida que puede usar para servir archivos localmente en desarrollo.

```
views.serve(request, path)
```

Esta función de visualización sirve para los archivos estáticos en desarrollo. La vista sólo funcionará si `DEBUG` es `True`. Esto se debe a que esta visión es extremadamente ineficiente, y probablemente insegura. Esto solo está destinado al desarrollo local, y nunca debe usarse en la producción.