HINTS

PERMISOS DE ADMINISTRADOR Y MODELO DE DJANGO

El administrador de Django tiene una integración muy estrecha con el sistema de autenticación integrado y, en particular, con los permisos del modelo. Fuera de la caja, el administrador está aplicando los permisos del modelo:

- Si el usuario no tiene permisos en un modelo, no podrá verlo ni acceder a él en el administrador.
- Si el usuario tiene permisos para ver y cambiar un modelo, podrá ver y actualizar las instancias, pero no podrá agregar nuevas instancias ni eliminar las existentes.

Con los permisos adecuados, es menos probable que los usuarios administradores cometan errores, y los intrusos tendrán más dificultades para causar daño.

IMPEDIR CONDICIONALMENTE LA ACTUALIZACIÓN DE CAMPOS

A veces es útil actualizar los campos directamente en el administrador. Pero no desea permitir que ningún usuario lo haga: desea permitir que solo los superusuario lo hagan. Supongamos que desea evitar que los usuarios que no son superusuarios cambien el nombre de usuario de un usuario. Para hacerlo, debe modificar el formulario de cambio generado por Django, y deshabilitar el campo de nombre de usuario según el usuario actual:

```
from django.contrib import admin
from django.contrib.auth.models import User
from django.contrib.auth.admin import UserAdmin

@admin.register(User)
class CustomUserAdmin(UserAdmin):
    def get_form(self, request, obj=None, **kwargs):
        form = super().get_form(request, obj, **kwargs)
        is_superuser = request.user.is_superuser

        if not is_superuser:
            form.base_fields['username'].disabled = True

        return form
```

Para realizar ajustes en el formulario, anule `get_form()`. Django utiliza esta función para generar un formulario de cambio predeterminado para un modelo. Para deshabilitar condicionalmente el campo,

primero debe buscar el formulario predeterminado generado por Django, y luego, si el usuario no es un super usuario, deshabilitar el campo de nombre de usuario.

Ahora, cuando alguien que no sea superusuario intente hacer una edición, el campo de nombre de usuario se desactivará. Cualquier intento de modificar el nombre de usuario a través de Django Admin fallará. Al contrario, cuando un superusuario intenta hacerlo, el campo de nombre de usuario será editable y se comportará como se esperaba.

IMPEDIR LA ACTUALIZACIÓN DE CAMPOS

Los formularios de administración desatendidos son los principales candidatos para errores. Un usuario del personal puede actualizar fácilmente una instancia de modelo a través del administrador, esto de una manera que la aplicación no espera. La mayoría de las veces, el usuario ni siquiera notará que algo anda mal, por lo que dichos errores suelen ser muy difíciles de rastrear y corregir.

Para evitar que ocurran, se puede restringir que los usuarios administradores modifiquen ciertos campos en el modelo. Si desea evitar que cualquier usuario, incluidos los superusuarios, actualice un campo, puede marcarlo como de solo lectura. Por ejemplo: el campo `date_joined` se establece cuando un usuario se registra. Esta información nunca debe ser modificada por ningún usuario, por lo que la marcas como de solo lectura:

```
from django.contrib import admin
from django.contrib.auth.models import User
from django.contrib.auth.admin import UserAdmin

@admin.register(User)
class CustomUserAdmin(UserAdmin):
    readonly_fields = [
        'date_joined',
    ]
```

Cuando se agrega un campo a `readonly_fields`, no se podrá editar en el formulario de cambios predeterminado del administrador. Cuando un campo se marca como de solo lectura, Django mostrará el elemento de entrada como deshabilitado.

CONSEJOS CONCEPTUALES

El script `manage.py` se usa para crear aplicaciones, trabajar con bases de datos, y empezar el desarrollo del servidor web. Algunos ejemplos del uso de `manage.py` son:

```
$ python manage.py check app1
$ python manage.py diffsettings
$ python manage.py flush
$ python manage.py makemigrations app1 app2
```

Si desea iniciar y volver a ejecutar todas las migraciones desde una base de datos vacía, debe eliminar y volver a crear la base de datos, y luego ejecutar `migrate`, el cual eliminará las tablas de datos originales.

Al cancelar la aplicación de migraciones, todas las dependientes también se cancelarán, independientemente de `<app_label>`. Puede usar --plan para verificar qué migraciones no se aplicarán. Si se ejecuta `python manage.py runserver` como un usuario con privilegios normales, que es lo más recomendado, es posible que no tenga acceso para iniciar un puerto en un número de puerto bajo. Los números de puerto bajos están reservados para el superusuario.

El servidor de desarrollo recarga automáticamente el código de Python para cada solicitud, según sea necesario. No necesita reiniciar el servidor para que los cambios en el código surtan efecto. Sin embargo, algunas acciones tales como agregar archivos no desencadenan un reinicio, por lo que en estos casos se deberá reiniciar el servidor. El comando `dbshell` asume que los programas están en su ruta, de modo que una llamada al nombre del programa (psql, mysql, etc) lo encontrará en el lugar correcto. No hay forma de especificar la ubicación del programa manualmente.

Al utilizar el comando `startapp`, si solo se proporciona el nombre de la aplicación, el directorio de la aplicación se creará en el directorio de trabajo actual. Si se proporciona el destino opcional, Django lo utilizará en lugar de crear uno nuevo. Puede usar `'.'` para indicar el directorio de trabajo actual.

Al crear un proyecto con `startproject`, si solo se proporciona el nombre del proyecto, tanto el directorio del proyecto como el paquete del proyecto se llamarán `<nombre del proyecto>`, y el directorio del proyecto se creará en el directorio de trabajo actual. Si se proporciona el destino opcional, Django usará ese directorio existente como el directorio del proyecto, y creará `manage.py` y el paquete del proyecto dentro de él. Use '.' para indicar el directorio de trabajo actual. El comando

`createsuperuser` solo está disponible si está instalado el sistema de autenticación de Django (`django.contrib.auth`).

Nunca ejecute `` `DEBUG` `` en producción. Si éste se establece en `True` en el archivo de configuración, los errores se mostrarán con rastreos completos. Estos rastreos normalmente contienen información que no deben ver los usuarios finales. También es posible que se tengan otras configuraciones o métodos que solo estén habilitados en el modo de depuración, y que podrían representar un riesgo para sus usuarios y sus datos. Para evitarlo, es importante usar diferentes archivos de configuración para el desarrollo local y para la implementación en producción.

El administrador debe indicar explícitamente en qué entorno nos encontramos para evitar que los usuarios eliminen accidentalmente los datos de producción. Esto se puede lograr usando la biblioteca `django-admin-env-notice`, donde se colocará un banner codificado por colores en la parte superior del sitio de administración. Una buena práctica para asegurar una aplicación es buscar cualquier error de seguridad, así:

```
python manage.py check -deploy
```

Si se ejecuta este comando localmente, se pueden detectar algunas advertencias que no serán relevantes en producción. Es posible crear usuarios en un sistema a través de un script de la siguiente manera:

```
from django.contrib.auth.models import User

# Create user and save to the database

user = User.objects.create_user('myusername', 'myemail@mail.com',
'mypassword')

# Update fields and then save again
user.first_name = 'Peter'
user.last_name = 'Glup'
user.save()
```