# 19 Cuentas de usuario



En el corazón de una aplicación web está la posibilidad de que cualquier usuario, en cualquier parte del mundo, pueda registrar una cuenta en tu aplicación y empezar a utilizarla. En este capítulo, construirás formularios para que los usuarios puedan añadir sus propios temas y entradas, y editar las entradas existentes. También aprenderás cómo Django protege contra ataques comunes a las páginas basadas en formularios, para que no tengas que dedicar mucho tiempo a pensar en la seguridad de tus aplicaciones.

También implementarás un sistema de autenticación de usuarios. Construirás una página de registro para que los usuarios creen cuentas, y luego restringirás el acceso a ciertas páginas sólo a los usuarios registrados. Luego modificarás algunas de las funciones de vista para que los usuarios sólo puedan ver sus propios datos. Aprenderás a mantener seguros los datos de tus usuarios.

## Permitir que los usuarios introduzcan datos

Antes de construir un sistema de autenticación para crear cuentas, primero añadiremos algunas páginas que permitan a los usuarios introducir sus propios datos. Daremos a los usuarios la posibilidad de añadir un nuevo tema, añadir una nueva entrada y editar sus entradas anteriores.

Actualmente, sólo un superusuario puede introducir datos a través del sitio de administración. No queremos que los usuarios interactúen con el sitio de administración, así que utilizaremos las herramientas de creación de formularios de Django para crear páginas que permitan a los usuarios introducir datos.

### Añadir nuevos temas

Empecemos por permitir a los usuarios añadir un nuevo tema. Añadir una página basada en un formulario funciona de forma muy parecida a añadir las páginas que ya hemos construido: definimos una URL, escribimos una función de vista y escribimos una plantilla. La única diferencia significativa es la adición de un nuevo módulo llamado *forms.py*, que contendrá los formularios.

#### El Tema ModeloFormulario

Cualquier página que permita a un usuario introducir y enviar información en una página web implica un elemento HTML llamado *form*. Cuando los usuarios introducen información, necesitamos *validate* que la información proporcionada sea el tipo correcto de datos y no sea maliciosa, como código diseñado para interrumpir nuestro servidor. A continuación, necesitamos procesar y guardar la información válida en el lugar adecuado de la base de datos. Django automatiza gran parte de este trabajo.

La forma más sencilla de construir un formulario en Django es utilizar un ModelForm, que utiliza la información de los modelos que definimos en el Capítulo 18 para construir un formulario automáticamente. Escribe tu primer formulario en el archivo *forms.py*, que debe crearse en el mismo directorio que *models.py*:

**forms.py**

from django import forms  
  
from .models import Topic  
  
❶ class TopicForm(forms.ModelForm):  
 class Meta:  
❷ model = Topic  
❸ fields = ['text']  
❹ labels = {'text': ''}

Primero importamos el módulo forms y el modelo con el que trabajaremos, Topic. A continuación, definimos una clase llamada TopicForm, que hereda de forms.ModelForm ❶.

La versión más sencilla de ModelForm consiste en una clase anidada Meta que indica a Django en qué modelo debe basarse el formulario y qué campos debe incluir. Aquí especificamos que el formulario debe basarse en el modelo Topic ❷, y que sólo debe incluir el campo text ❸. La cadena vacía en el diccionario de etiquetas indica a Django que no genere una etiqueta para el campo text ❹.

#### La URL del nuevo\_tema

La URL de una nueva página debe ser corta y descriptiva. Cuando el usuario quiera añadir un nuevo tema, le enviaremos a *http://localhost:8000/new\_topic/*. Éste es el patrón de URL para la página new\_topic; añádelo a *learning\_logs/urls.py*:

**learning\_logs/urls.py**

--snip--  
urlpatterns = [  
 --snip--  
 # Page for adding a new topic.  
 path('new\_topic/', views.new\_topic, name='new\_topic'),  
]

Este patrón de URL envía peticiones a la función de vista new\_topic(), que escribiremos a continuación.

#### La función de vista new\_topic()

La función new\_topic() debe gestionar dos situaciones diferentes: las solicitudes iniciales de la página new\_topic, en cuyo caso debe mostrar un formulario en blanco; y el procesamiento de los datos enviados en el formulario. Una vez procesados los datos de un formulario enviado, debe redirigir al usuario de nuevo a la página topics:

**views.py**

from django.shortcuts import render, redirect  
  
from .models import Topic  
from .forms import TopicForm  
  
--snip--  
def new\_topic(request):  
 """Add a new topic."""  
❶ if request.method != 'POST':  
 # No data submitted; create a blank form.  
❷ form = TopicForm()  
 else:  
 # POST data submitted; process data.  
❸ form = TopicForm(data=request.POST)  
❹ if form.is\_valid():  
❺ form.save()  
❻ return redirect('learning\_logs:topics')  
  
 # Display a blank or invalid form.  
❼ context = {'form': form}  
 return render(request, 'learning\_logs/new\_topic.html', context)

Importamos la función redirect, que utilizaremos para redirigir al usuario de vuelta a la página topics después de que envíe su tema. También importamos el formulario que acabamos de escribir, TopicForm.

#### Peticiones GET y POST

Los dos tipos principales de peticiones que utilizarás cuando construyas aplicaciones son GET y POST. Las peticiones *GET* se utilizan para páginas que sólo leen datos del servidor. Normalmente utilizas las peticiones *POST* cuando el usuario necesita enviar información a través de un formulario. Especificaremos el método POST para procesar todos nuestros formularios. (Existen algunos otros tipos de peticiones, pero no los utilizaremos en este proyecto).

La función new\_topic() toma como parámetro el objeto request. Cuando el usuario solicite inicialmente esta página, su navegador enviará una solicitud GET. Una vez que el usuario haya rellenado y enviado el formulario, su navegador enviará una solicitud POST. Dependiendo de la petición, sabremos si el usuario está solicitando un formulario en blanco (GET) o pidiéndonos que procesemos un formulario cumplimentado (POST).

Utilizamos una prueba if para determinar si el método de solicitud es GET o POST ❶. Si el método de solicitud no es POST, probablemente la solicitud sea GET, por lo que tenemos que devolver un formulario en blanco. (Si es otro tipo de solicitud, sigue siendo seguro devolver un formulario en blanco.) Creamos una instancia de TopicForm ❷, la asignamos a la variable form, y enviamos el formulario a la plantilla del diccionario context ❼. Como no incluimos argumentos al instanciar TopicForm, Django crea un formulario en blanco que el usuario puede rellenar.

Si el método de solicitud es POST, el bloque else se ejecuta y procesa los datos enviados en el formulario. Creamos una instancia de TopicForm ❸ y le pasamos los datos introducidos por el usuario, que se asignan a request.POST. El objeto form que se devuelve contiene la información enviada por el usuario.

No podemos guardar la información enviada en la base de datos hasta que hayamos comprobado que es válida ❹. El método is\_valid() comprueba que se hayan rellenado todos los campos obligatorios (todos los campos de un formulario son obligatorios por defecto) y que los datos introducidos coincidan con los tipos de campo esperados; por ejemplo, que la longitud de text sea inferior a 200 caracteres, como especificamos en *models.py* en el Capítulo 18. Esta validación automática nos ahorra mucho trabajo. Si todo es válido, podemos llamar a save() ❺, que escribe los datos del formulario en la base de datos.

Una vez guardados los datos, podemos abandonar esta página. La función redirect() toma el nombre de una vista y redirige al usuario a la página asociada a esa vista. Aquí utilizamos redirect() para redirigir el navegador del usuario a la página topics ❻, donde el usuario debería ver el tema que acaba de introducir en la lista de temas.

La variable context se define al final de la función de vista, y la página se renderiza utilizando la plantilla *new\_topic.html*, que crearemos a continuación. Este código se coloca fuera de cualquier bloque if; se ejecutará si se ha creado un formulario en blanco, y se ejecutará si se determina que un formulario enviado no es válido. Un formulario inválido incluirá algunos mensajes de error por defecto para ayudar al usuario a enviar datos aceptables.

#### La plantilla new\_topic

Ahora crearemos una nueva plantilla llamada *new\_topic.html* para mostrar el formulario que acabamos de crear:

**new\_topic.html**

{% extends "learning\_logs/base.html" %}  
  
{% block content %}  
 <p>Add a new topic:</p>  
  
❶ <form action="{% url 'learning\_logs:new\_topic' %}" method='post'>  
❷ {% csrf\_token %}  
❸ {{ form.as\_div }}  
❹ <button name="submit">Add topic</button>  
 </form>  
  
{% endblock content %}

Esta plantilla extiende *base.html*, por lo que tiene la misma estructura base que el resto de páginas del Registro de Aprendizaje. Utilizamos las etiquetas <form></form> para definir un formulario HTML ❶. El argumento action indica al navegador dónde enviar los datos enviados en el formulario; en este caso, los enviamos a la función de vista new\_topic(). El argumento method indica al navegador que envíe los datos como una petición POST.

Django utiliza la etiqueta de plantilla {% csrf\_token %} ❷ para evitar que los atacantes utilicen el formulario para obtener acceso no autorizado al servidor. (Este tipo de ataque se denomina *cross-site request forgery*.) A continuación, mostramos el formulario; aquí puedes ver lo sencillas que Django puede hacer ciertas tareas, como mostrar un formulario. Sólo tenemos que incluir la variable de plantilla {{ form.as\_div }} para que Django cree automáticamente todos los campos necesarios para mostrar el formulario ❸. El modificador as\_div indica a Django que muestre todos los elementos del formulario como elementos HTML <div></div>; ésta es una forma sencilla de mostrar el formulario de forma ordenada.

Django no crea un botón de envío para los formularios, así que definimos uno antes de cerrar el formulario ❹.

#### Enlace a la página nuevo\_tema

A continuación, incluimos un enlace a la página new\_topic en la página topics:

**topics.html**

{% extends "learning\_logs/base.html" %}  
  
{% block content %}  
  
 <p>Topics</p>  
  
 <ul>  
 --snip--  
 </ul>  
  
 <a href="{% url 'learning\_logs:new\_topic' %}">Add a new topic</a>  
  
{% endblock content %}

Coloca el enlace después de la lista de temas existentes. [La Figura 19-1](#figure19-1) muestra el formulario resultante; prueba a utilizarlo para añadir algunos temas nuevos por tu cuenta.



Figura 19-1: Página para añadir un nuevo

tema

### Añadir nuevos temas

Ahora que el usuario puede añadir un nuevo tema, también querrá añadir nuevas entradas. Definiremos de nuevo una URL, escribiremos una función de vista y una plantilla, y enlazaremos con la página. Pero antes, añadiremos otra clase a *forms.py*.

#### El modelo de formulario de entrada

Necesitamos crear un formulario asociado al modelo Entry, pero esta vez, con un poco más de personalización que TopicForm:

**forms.py**

from django import forms  
  
from .models import Topic, Entry  
  
class TopicForm(forms.ModelForm):  
 --snip--  
  
class EntryForm(forms.ModelForm):  
 class Meta:  
 model = Entry  
 fields = ['text']  
❶ labels = {'text': ''}  
❷ widgets = {'text': forms.Textarea(attrs={'cols': 80})}

Actualizamos la declaración import para incluir tanto Entry como Topic. Creamos una nueva clase llamada EntryForm que hereda de forms.ModelForm. La clase EntryForm tiene una clase anidada Meta que enumera el modelo en el que se basa y el campo a incluir en el formulario. Volvemos a dar al campo 'text' una etiqueta en blanco ❶.

Para EntryForm, incluimos el atributo widgets ❷. Un *widget* es un elemento de formulario HTML, como un cuadro de texto de una línea, un área de texto multilínea o una lista desplegable. Al incluir el atributo widgets, puedes anular las opciones de widget predeterminadas de Django . Aquí le estamos diciendo a Django que utilice un elemento forms.Textarea con una anchura de 80 columnas, en lugar de las 40 columnas por defecto. Esto da a los usuarios espacio suficiente para escribir una entrada con sentido.

#### La URL de la nueva\_entrada

Las nuevas entradas deben asociarse a un tema concreto, por lo que necesitamos incluir un argumento topic\_id en la URL para añadir una nueva entrada. Aquí tienes la URL, que debes añadir a *learning\_logs/urls.py*:

**learning\_logs/urls.py**

--snip--  
urlpatterns = [  
 --snip--  
 # Page for adding a new entry.  
 path('new\_entry/<int:topic\_id>/', views.new\_entry, name='new\_entry'),  
]

Este patrón de URL coincide con cualquier URL de la forma *http://localhost:8000/new\_entry/id/*, donde id es un número que coincide con el ID del tema. El código <int:topic\_id> captura un valor numérico y lo asigna a la variable topic\_id. Cuando se solicita una URL que coincide con este patrón, Django envía la solicitud y el ID del tema a la función de vista new\_entry().

#### La función de vista new\_entry()

La función de vista para new\_entry es muy parecida a la función para añadir un nuevo tema. Añade el siguiente código a tu archivo *views.py*:

**views.py**

from django.shortcuts import render, redirect  
  
from .models import Topic  
from .forms import TopicForm, EntryForm  
  
--snip--  
def new\_entry(request, topic\_id):  
 """Add a new entry for a particular topic."""  
❶ topic = Topic.objects.get(id=topic\_id)  
  
❷ if request.method != 'POST':  
 # No data submitted; create a blank form.  
❸ form = EntryForm()  
 else:  
 # POST data submitted; process data.  
❹ form = EntryForm(data=request.POST)  
 if form.is\_valid():  
❺ new\_entry = form.save(commit=False)  
❻ new\_entry.topic = topic  
 new\_entry.save()  
❼ return redirect('learning\_logs:topic', topic\_id=topic\_id)  
  
 # Display a blank or invalid form.  
 context = {'topic': topic, 'form': form}  
 return render(request, 'learning\_logs/new\_entry.html', context)

Actualizamos la declaración import para incluir el EntryForm que acabamos de hacer. La definición de new\_entry() tiene un parámetro topic\_id para almacenar el valor que recibe de la URL. Necesitaremos el tema para renderizar la página y procesar los datos del formulario, así que utilizamos topic\_id para obtener el objeto tema ❶ correcto .

A continuación, comprobamos si el método de solicitud es POST o GET ❷. El bloque if se ejecuta si es una petición GET, y creamos una instancia vacía de EntryForm ❸.

Si el método de solicitud es POST, procesamos los datos creando una instancia de EntryForm, rellenada con los datos POST del objeto request ❹. A continuación, comprobamos si el formulario es válido. Si lo es, tenemos que establecer el atributo topic del objeto de entrada antes de guardarlo en la base de datos. Cuando llamamos a save(), incluimos el argumento commit=False ❺ para decirle a Django que cree un nuevo objeto de entrada y lo asigne a new\_entry, sin guardarlo aún en la base de datos. Establecemos el atributo topic de new\_entry al tema que sacamos de la base de datos al principio de la función ❻. A continuación, llamamos a save() sin argumentos, guardando la entrada en la base de datos con el tema asociado correcto.

La llamada a redirect() requiere dos argumentos: el nombre de la vista a la que queremos redirigir y el argumento que requiere la función vista ❼. Aquí, estamos redirigiendo a topic(), que necesita el argumento topic\_id. Esta vista muestra la página del tema para el que el usuario hizo una entrada, y debería ver su nueva entrada en la lista de entradas.

Al final de la función, creamos un diccionario context y mostramos la página utilizando la plantilla *new\_entry.html*. Este código se ejecutará para un formulario en blanco, o para un formulario que ha sido enviado pero resulta ser inválido.

#### La plantilla new\_entry

Como puedes ver en el código siguiente, la plantilla para new\_entry es similar a la plantilla para new\_topic:

**new\_entry.html**

{% extends "learning\_logs/base.html" %}  
  
{% block content %}  
  
❶ <p><a href="{% url 'learning\_logs:topic' topic.id %}">{{ topic }}</a></p>  
  
 <p>Add a new entry:</p>  
❷ <form action="{% url 'learning\_logs:new\_entry' topic.id %}" method='post'>  
 {% csrf\_token %}  
 {{ form.as\_div }}  
 <button name='submit'>Add entry</button>  
 </form>  
  
{% endblock content %}

Mostramos el tema en la parte superior de la página ❶, para que el usuario pueda ver a qué tema está añadiendo una entrada. El tema también actúa como enlace a la página principal de ese tema.

El argumento action del formulario incluye el valor topic.id en la URL, para que la función de vista pueda asociar la nueva entrada con el tema ❷ correcto . Aparte de eso, esta plantilla tiene el mismo aspecto que *new\_topic.html*.

#### Enlace a la página nueva\_entrada

A continuación, tenemos que incluir un enlace a la página new\_entry desde cada página de tema, en la plantilla de tema:

**topic.html**

{% extends "learning\_logs/base.html" %}  
  
{% block content %}  
  
 <p>Topic: {{ topic }}</p>  
  
 <p>Entries:</p>  
 <p>  
 <a href="{% url 'learning\_logs:new\_entry' topic.id %}">Add new entry</a>  
 </p>  
  
 <ul>  
 --snip--  
 </ul>  
  
{% endblock content %}

Colocamos el enlace para añadir entradas justo antes de mostrar las entradas, porque añadir una nueva entrada será la acción más común en esta página. [La Figura 19-2](#figure19-2) muestra la página new\_entry. Ahora los usuarios pueden añadir nuevos temas y tantas entradas como quieran para cada tema. Prueba la página new\_entry añadiendo algunas entradas a algunos de los temas que has creado.

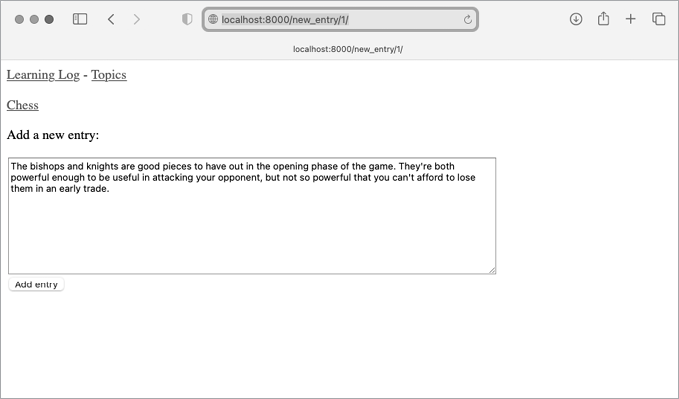


Figura 19-2: La

página



new\_entry

### Editar entradas

Ahora crearemos una página para que los usuarios puedan editar las entradas que han añadido.

#### La URL editar\_entrada

La URL de la página tiene que pasar el ID de la entrada que se va a editar. Aquí tienes *learning\_logs/urls.py*:

**urls.py**

--snip--  
urlpatterns = [  
 --snip--  
 # Page for editing an entry.  
 path('edit\_entry/<int:entry\_id>/', views.edit\_entry, name='edit\_entry'),  
]

Este patrón de URL coincide con URL como *http://localhost:8000/edit\_entry/id/*. Aquí el valor de id se asigna al parámetro entry\_id. Django envía las solicitudes que coinciden con este formato a la función de vista edit\_entry().

#### La función de vista edit\_entry()

Cuando la página edit\_entry recibe una solicitud GET, la función edit\_entry() devuelve un formulario para editar la entrada. Cuando la página recibe una petición POST con el texto de la entrada revisado, guarda el texto modificado en la base de datos:

**views.py**

from django.shortcuts import render, redirect  
  
from .models import Topic, Entry  
from .forms import TopicForm, EntryForm  
--snip--  
  
def edit\_entry(request, entry\_id):  
 """Edit an existing entry."""  
❶ entry = Entry.objects.get(id=entry\_id)  
 topic = entry.topic  
  
 if request.method != 'POST':  
 # Initial request; pre-fill form with the current entry.  
❷ form = EntryForm(instance=entry)  
 else:  
 # POST data submitted; process data.  
❸ form = EntryForm(instance=entry, data=request.POST)  
 if form.is\_valid():  
❹ form.save()  
❺ return redirect('learning\_logs:topic', topic\_id=topic.id)  
  
 context = {'entry': entry, 'topic': topic, 'form': form}  
 return render(request, 'learning\_logs/edit\_entry.html', context)

Primero importamos el modelo Entry. A continuación, obtenemos el objeto entrada que el usuario quiere editar ❶ y el tema asociado a esta entrada. En el bloque if  , que se ejecuta para una solicitud GET, creamos una instancia de EntryForm con el argumento instance=entry ❷. Este argumento indica a Django que cree el formulario, rellenado previamente con la información del objeto de entrada existente. El usuario verá sus datos existentes y podrá editarlos.

Al procesar una solicitud POST, pasamos los argumentos instance=entry y data=request.POST ❸. Estos argumentos indican a Django que cree una instancia de formulario basada en la información asociada al objeto de entrada existente, actualizada con cualquier dato relevante de request.POST. A continuación, comprobamos si el formulario es válido; si lo es, llamamos a save() sin argumentos porque la entrada ya está asociada al tema ❹ correcto . A continuación, redirigimos a la página topic, donde el usuario debería ver la versión actualizada de la entrada que editó ❺.

Si mostramos un formulario inicial para editar la entrada o si el formulario enviado no es válido, creamos el diccionario context y renderizamos la página utilizando la plantilla *edit\_entry.html*.

#### La plantilla edit\_entry

A continuación, creamos una plantilla *edit\_entry.html* , que es similar a *new\_entry.html*:

**edit\_entry.html**

{% extends "learning\_logs/base.html" %}  
  
{% block content %}  
  
 <p><a href="{% url 'learning\_logs:topic' topic.id %}">{{ topic }}</a></p>  
  
 <p>Edit entry:</p>  
  
❶ <form action="{% url 'learning\_logs:edit\_entry' entry.id %}" method='post'>  
 {% csrf\_token %}  
 {{ form.as\_div }}  
❷ <button name="submit">Save changes</button>  
 </form>  
  
{% endblock content %}

El argumento action devuelve el formulario a la función edit\_entry() para su procesamiento ❶. Incluimos el entry.id como argumento en la etiqueta {% url %}, para que la función vista pueda modificar el objeto de entrada correcto. Etiquetamos el botón de envío como Save changes para recordar al usuario que está guardando ediciones, no creando una nueva entrada ❷.

#### Enlace a la página edit\_entry

Ahora tenemos que incluir un enlace a la página edit\_entry para cada entrada de la página de temas:

**topic.html**

--snip--  
 {% for entry in entries %}  
 <li>  
 <p>{{ entry.date\_added|date:'M d, Y H:i' }}</p>  
 <p>{{ entry.text|linebreaks }}</p>  
 <p>  
 <a href="{% url 'learning\_logs:edit\_entry' entry.id %}">  
 Edit entry</a></p>  
 </li>  
--snip--

Incluimos el enlace de edición después de mostrar la fecha y el texto de cada entrada. Utilizamos la etiqueta de plantilla {% url %} para determinar la URL del patrón de URL con nombre edit\_entry, junto con el atributo ID de la entrada actual en el bucle (entry.id). El texto del enlace Edit entry aparece después de cada entrada de la página. [La Figura 19-3](#figure19-3) muestra el aspecto de la página del tema con estos enlaces.

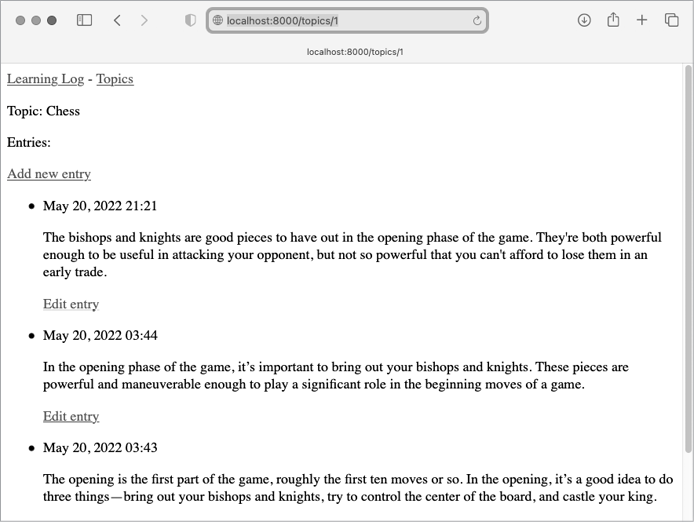


Figura 19-3: Cada entrada tiene ahora un enlace para editarla.

El Registro de Aprendizaje tiene ahora la mayor parte de la funcionalidad que necesita. Los usuarios pueden añadir temas y entradas, y pueden leer cualquier conjunto de entradas que deseen. En la próxima sección, implementaremos un sistema de registro de usuarios para que cualquiera pueda crear una cuenta en Learning Log y crear su propio conjunto de temas y entradas.

## Pruébalo tú mismo

19-1. Blog: Inicia un nuevo proyecto Django llamado *Blog*. Crea una aplicación llamada *blogs*, con un modelo que represente un blog en general, y un modelo que represente una entrada individual del blog. Asigna a cada modelo un conjunto adecuado de campos. Crea un superusuario para el proyecto y utiliza el sitio de administración para crear un blog y un par de entradas cortas. Crea una página de inicio que muestre todas las entradas en el orden adecuado.

Crea páginas para crear un blog, para crear nuevas entradas y para editar las entradas existentes. Utiliza tus páginas para asegurarte de que funcionan.

## Configurar cuentas de usuario

En esta sección, configuraremos un sistema de registro y autorización de usuarios para que la gente pueda registrar una cuenta, entrar y salir. Crearemos una nueva app que contenga toda la funcionalidad relacionada con el trabajo con usuarios. Utilizaremos el sistema de autenticación de usuarios por defecto incluido en Django para realizar la mayor parte del trabajo posible. También modificaremos ligeramente el modelo Topic para que cada tema pertenezca a un usuario determinado.

### La App de cuentas

Empezaremos creando una nueva app llamada accounts, utilizando el comando startapp:

(ll\_env)learning\_log$ python manage.py startapp accounts  
(ll\_env)learning\_log$ ls  
❶ accounts db.sqlite3 learning\_logs ll\_env ll\_project manage.py  
(ll\_env)learning\_log$ ls accounts  
❷ \_\_init\_\_.py admin.py apps.py migrations models.py tests.py views.py

El sistema de autenticación por defecto está construido en torno al concepto de cuentas de usuario, por lo que utilizar el nombre accounts facilita la integración con el sistema por defecto. El comando startapp que se muestra aquí crea un nuevo directorio llamado *accounts* ❶ con una estructura idéntica a la de la app learning\_logs ❷.

#### Añadir cuentas a settings.py

Tenemos que añadir nuestra nueva app a INSTALLED\_APPS en *settings.py*, de la siguiente manera:

**settings.py**

--snip--  
INSTALLED\_APPS = [  
 # My apps  
 'learning\_logs',  
 'accounts',  
  
 # Default django apps.  
 --snip--  
]  
--snip--

Ahora Django incluirá la app accounts en el proyecto global.

#### Incluir las URL de las cuentas

A continuación, tenemos que modificar la raíz *urls.py* para que incluya las URL que escribiremos para la app accounts:

**ll\_project/urls.py**

from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('accounts/', include('accounts.urls')),  
 path('', include('learning\_logs.urls')),  
]

Añadimos una línea para incluir el archivo *urls.py* de accounts. Esta línea coincidirá con cualquier URL que empiece por la palabra *accounts*, como *http://localhost:8000/accounts/login/*.

### La página de inicio de sesión

Primero implementaremos una página de inicio de sesión. Utilizaremos la vista login predeterminada que proporciona Django, por lo que el patrón de URL para esta aplicación es un poco diferente. Crea un nuevo archivo *urls.py* en el directorio *ll\_project/accounts/* y añádele lo siguiente:

**accounts/urls.py**

"""Defines URL patterns for accounts."""  
  
from django.urls import path, include  
  
app\_name = 'accounts'  
urlpatterns = [  
 # Include default auth urls.  
 path('', include('django.contrib.auth.urls')),  
]

Importamos la función path, y luego importamos la función include para poder incluir algunas URL de autenticación por defecto que Django ha definido. Estas URL predeterminadas incluyen patrones de URL con nombre, como 'login' y 'logout'. Establecemos la variable app\_name en 'accounts' para que Django pueda distinguir estas URL de las que pertenecen a otras aplicaciones. Incluso las URL predeterminadas proporcionadas por Django, cuando se incluyan en el archivo *urls.py* de la app accounts, serán accesibles a través del espacio de nombres accounts.

El patrón de la página de acceso coincide con la URL *http://localhost:8000/accounts/login/*. Cuando Django lee esta URL, la palabra *accounts* indica a Django que busque en *accounts/urls.py*, y *login* le indica que envíe las peticiones a la vista por defecto de Django login.

#### La plantilla de inicio de sesión

Cuando el usuario solicite la página de inicio de sesión, Django utilizará una función de vista por defecto, pero aún tenemos que proporcionar una plantilla para la página. Las vistas de autenticación por defecto buscan plantillas dentro de una carpeta llamada *registration*, así que tendremos que crear esa carpeta. Dentro del directorio *ll\_project/accounts/*, crea un directorio llamado *templates*; dentro de éste, crea otro directorio llamado *registration*. Aquí está la plantilla *login.html*, que debería guardarse en *ll\_project/accounts/templates/registration*:

**login.html**

{% extends 'learning\_logs/base.html' %}  
  
{% block content %}  
  
❶ {% if form.errors %}  
 <p>Your username and password didn't match. Please try again.</p>  
 {% endif %}  
  
❷ <form action="{% url 'accounts:login' %}" method='post'>  
 {% csrf\_token %}  
❸ {{ form.as\_div }}  
  
❹ <button name="submit">Log in</button>  
 </form>  
  
{% endblock content %}

Esta plantilla amplía *base.html* para garantizar que la página de inicio de sesión tenga el mismo aspecto que el resto del sitio. Ten en cuenta que una plantilla de una aplicación puede heredar de una plantilla de otra aplicación.

Si se establece el atributo errors del formulario, mostramos un mensaje de error ❶, informando de que la combinación de nombre de usuario y contraseña no coincide con nada almacenado en la base de datos.

Queremos que la vista de inicio de sesión procese el formulario, así que establecemos el argumento action como la URL de la página de inicio de sesión ❷. La vista de inicio de sesión envía un objeto form a la plantilla, y nos toca a nosotros mostrar el formulario ❸ y añadir un botón de envío ❹.

#### El ajuste LOGIN\_REDIRECT\_URL

Una vez que un usuario se registra correctamente, Django necesita saber dónde enviarlo. Esto lo controlamos en el archivo de configuración.

Añade el siguiente código al final de *settings.py*:

**settings.py**

--snip--  
# My settings.  
LOGIN\_REDIRECT\_URL = 'learning\_logs:index'

Con todos los ajustes por defecto en *settings.py*, es útil marcar la sección en la que vamos a añadir nuevos ajustes. La primera configuración nueva que añadiremos es LOGIN\_REDIRECT\_URL, que indica a Django a qué URL debe redirigir después de un intento de inicio de sesión con éxito.

#### Enlace a la página de inicio de sesión

Vamos a añadir el enlace de inicio de sesión a *base.html* para que aparezca en todas las páginas. No queremos que el enlace aparezca cuando el usuario ya haya iniciado sesión, así que lo anidaremos dentro de una etiqueta {% if %}:

**base.html**

<p>  
 <a href="{% url 'learning\_logs:index' %}">Learning Log</a> -  
 <a href="{% url 'learning\_logs:topics' %}">Topics</a> -  
❶ {% if user.is\_authenticated %}  
❷ Hello, {{ user.username }}.  
 {% else %}  
❸ <a href="{% url 'accounts:login' %}">Log in</a>  
 {% endif %}  
</p>  
  
{% block content %}{% endblock content %}

En el sistema de autenticación de Django, cada plantilla tiene un objeto user disponible que siempre tiene un atributo is\_authenticated establecido: el atributo es True si el usuario está conectado y False si no lo está. Este atributo te permite mostrar un mensaje a los usuarios autenticados y otro a los no autenticados.

Aquí mostramos un saludo a los usuarios conectados ❶. Los usuarios autenticados tienen un atributo adicional username, que utilizamos para personalizar el saludo y recordar al usuario que está conectado ❷. Para los usuarios que no se han autenticado, mostramos un enlace a la página de inicio de sesión ❸.

#### Utilizar la página de inicio de sesión

Ya hemos configurado una cuenta de usuario, así que vamos a iniciar sesión para ver si la página funciona. Ve a *http://localhost:8000/admin/*. Si todavía has iniciado sesión como administrador, busca un enlace de **cierre de sesión** en la cabecera y haz clic en él.

Cuando hayas cerrado la sesión, ve a *http://localhost:8000/accounts/login/*. Deberías ver una página de inicio de sesión similar a la que se muestra en [la Figura 19-4](#figure19-4). Introduce el nombre de usuario y la contraseña que configuraste anteriormente, y deberías volver a la página de inicio. La cabecera de la página de inicio debería mostrar un saludo personalizado con tu nombre de usuario.

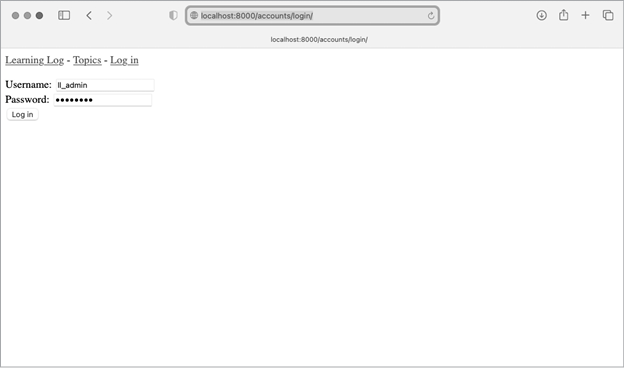


Figura 19-4: Página de inicio de

sesión

### Cerrar sesión

Ahora tenemos que proporcionar a los usuarios una forma de cerrar la sesión. Las solicitudes de cierre de sesión deben enviarse como solicitudes POST, así que añadiremos un pequeño formulario de cierre de sesión a *base.html*. Cuando los usuarios pulsen el botón de cierre de sesión, irán a una página que confirme que han cerrado la sesión.

#### Añadir un formulario de cierre de sesión a base.html

Añadiremos el formulario de cierre de sesión a *base.html* para que esté disponible en todas las páginas. Lo incluiremos en otro bloque de if, para que sólo puedan verlo los usuarios que ya hayan iniciado sesión:

**base.html**

--snip--  
{% block content %}{% endblock content %}  
  
{% if user.is\_authenticated %}  
❶ <hr />  
❷ <form action="{% url 'accounts:logout' %}" method='post'>  
 {% csrf\_token %}  
 <button name='submit'>Log out</button>  
 </form>  
{% endif %}

El patrón de URL por defecto para cerrar la sesión es 'accounts/logout/'. Sin embargo, la solicitud debe enviarse como solicitud POST; de lo contrario, los atacantes pueden forzar fácilmente las solicitudes de cierre de sesión. Para que la solicitud de cierre de sesión utilice POST, definimos un formulario sencillo.

Colocamos el formulario en la parte inferior de la página, debajo de un elemento de regla horizontal (<hr />) ❶. Esta es una forma sencilla de mantener siempre el botón de cierre de sesión en una posición coherente debajo de cualquier otro contenido de la página. El propio formulario tiene la URL de cierre de sesión como argumento action, y 'post' como método de solicitud ❷. Todo formulario en Django necesita incluir el {% csrf\_token %}, incluso un formulario sencillo como éste. Este formulario está vacío excepto por el botón de envío.

#### El ajuste LOGOUT\_REDIRECT\_URL

Cuando el usuario hace clic en el botón de cierre de sesión, Django necesita saber dónde enviarlo. Controlamos este comportamiento en *settings.py*:

**settings.py**

--snip--  
# My settings.  
LOGIN\_REDIRECT\_URL = 'learning\_logs:index'  
LOGOUT\_REDIRECT\_URL = 'learning\_logs:index'

El ajuste LOGOUT\_REDIRECT\_URL que se muestra aquí indica a Django que redirija a los usuarios que han cerrado la sesión de vuelta a la página de inicio. Esta es una forma sencilla de confirmar que se han desconectado, porque ya no deberían ver su nombre de usuario después de desconectarse.

### La página de registro

A continuación, crearemos una página para que los nuevos usuarios puedan registrarse. Utilizaremos la página predeterminada de Django UserCreationForm, pero escribiremos nuestra propia función de vista y plantilla.

#### La URL de registro

El siguiente código proporciona el patrón de URL para la página de registro, que debe colocarse en *accounts/urls.py*:

**accounts/urls.py**

"""Defines URL patterns for accounts."""  
  
from django.urls import path, include  
  
from . import views  
  
app\_name = accounts  
urlpatterns = [  
 # Include default auth urls.  
 path('', include('django.contrib.auth.urls')),  
 # Registration page.  
 path('register/', views.register, name='register'),  
]

Importamos el módulo views de accounts, que necesitamos porque estamos escribiendo nuestra propia vista para la página de registro. El patrón para la página de registro coincide con la URL *http://localhost:8000/accounts/register/* y envía peticiones a la función register() que vamos a escribir.

#### La función de la vista register()

La función de vista register() tiene que mostrar un formulario de registro en blanco cuando se solicite la página de registro por primera vez, y luego procesar los formularios de registro completados cuando se envíen. Cuando un registro se realiza correctamente, la función también debe iniciar la sesión del nuevo usuario. Añade el siguiente código a *accounts/views.py*:

**accounts/views.py**

from django.shortcuts import render, redirect  
from django.contrib.auth import login  
from django.contrib.auth.forms import UserCreationForm  
  
def register(request):  
 """Register a new user."""  
 if request.method != 'POST':  
 # Display blank registration form.  
❶ form = UserCreationForm()  
 else:  
 # Process completed form.  
❷ form = UserCreationForm(data=request.POST)  
  
❸ if form.is\_valid():  
❹ new\_user = form.save()  
 # Log the user in and then redirect to home page.  
❺ login(request, new\_user)  
❻ return redirect('learning\_logs:index')  
  
 # Display a blank or invalid form.  
 context = {'form': form}  
 return render(request, 'registration/register.html', context)

Importamos las funciones render() y redirect(), y luego importamos la función login() para iniciar la sesión del usuario si su información de registro es correcta. También importamos la función por defecto UserCreationForm. En la función register(), comprobamos si estamos respondiendo a una solicitud POST. Si no es así, creamos una instancia de UserCreationForm sin datos iniciales ❶.

Si estamos respondiendo a una petición POST, creamos una instancia de UserCreationForm basada en los datos enviados ❷. Comprobamos que los datos son válidos ❸ -eneste caso, que el nombre de usuario tiene los caracteres adecuados, las contraseñas coinciden y el usuario no está intentando hacer nada malicioso en su envío.

Si los datos enviados son válidos, llamamos al método save() del formulario para guardar el nombre de usuario y el hash de la contraseña en la base de datos ❹. El método save() devuelve el objeto usuario recién creado, que asignamos a new\_user. Una vez guardada la información del usuario, iniciamos su sesión llamando a la función login() con los objetos request y new\_user ❺, lo que crea una sesión válida para el nuevo usuario. Por último, redirigimos al usuario a la página de inicio ❻, , donde un saludo personalizado en la cabecera le informa de que su registro se ha realizado correctamente.

Al final de la función, mostramos la página, que será un formulario en blanco o un formulario enviado que no es válido.

#### La plantilla de registro

Ahora crea una plantilla para la página de registro, que será similar a la página de inicio de sesión. Asegúrate de guardarla en el mismo directorio que *login.html*:

**register.html**

{% extends "learning\_logs/base.html" %}  
  
{% block content %}  
  
 <form action="{% url 'accounts:register' %}" method='post'>  
 {% csrf\_token %}  
 {{ form.as\_div }}  
  
 <button name="submit">Register</button>  
 </form>  
  
{% endblock content %}

Debería parecerse a las demás plantillas basadas en formularios que hemos estado escribiendo. Volvemos a utilizar el método as\_div para que Django muestre todos los campos del formulario adecuadamente, incluyendo cualquier mensaje de error si el formulario no se rellena correctamente.

#### Enlace a la página de registro

A continuación, añadiremos código para mostrar el enlace a la página de registro a cualquier usuario que no haya iniciado sesión:

**base.html**

--snip--  
 {% if user.is\_authenticated %}  
 Hello, {{ user.username }}.  
 {% else %}  
 <a href="{% url 'accounts:register' %}">Register</a> -  
 <a href="{% url 'accounts:login' %}">Log in</a>  
 {% endif %}  
--snip--

Ahora los usuarios que han iniciado sesión ven un saludo personalizado y un botón de cierre de sesión. Los usuarios que no han iniciado sesión ven un enlace de registro y un enlace de inicio de sesión. Prueba la página de registro creando varias cuentas de usuario con diferentes nombres de usuario.

En la siguiente sección, restringiremos algunas de las páginas para que sólo estén disponibles para los usuarios registrados, y nos aseguraremos de que cada tema pertenezca a un usuario concreto.

## Nota

El sistema de registro que hemos configurado permite a cualquiera crear cualquier número de cuentas para Learning Log. Algunos sistemas requieren que los usuarios confirmen su identidad mediante el envío de un correo electrónico de confirmación al que los usuarios deben responder. De este modo, el sistema genera menos cuentas de spam que el sistema sencillo que utilizamos aquí. Sin embargo, cuando estás aprendiendo a crear aplicaciones, es perfectamente apropiado practicar con un sistema sencillo de registro de usuarios como el que estamos utilizando.

## Pruébalo tú mismo

19-2. Cuentas de Blog: Añade un sistema de autenticación y registro de usuarios al proyecto Blog que iniciaste en el Ejercicio 19-1(página 415). Asegúrate de que los usuarios registrados ven su nombre de usuario en algún lugar de la pantalla y que los usuarios no registrados ven un enlace a la página de registro.

## Permitir que los usuarios sean propietarios de sus datos

Los usuarios deben poder introducir datos privados en sus registros de aprendizaje, así que crearemos un sistema para averiguar qué datos pertenecen a cada usuario. Luego restringiremos el acceso a determinadas páginas para que los usuarios sólo puedan trabajar con sus propios datos.

Modificaremos el modelo Topic para que cada tema pertenezca a un usuario concreto. Esto también se ocupará de las entradas, porque cada entrada pertenece a un tema concreto. Empezaremos restringiendo el acceso a determinadas páginas.

### Restringir el acceso con @login\_required

Django facilita la restricción de acceso a determinadas páginas mediante el decorador @login\_required. Recuerda del Capítulo 11 que un *decorator* es una directiva que se coloca justo antes de la definición de una función y que modifica el comportamiento de ésta. Veamos un ejemplo.

#### Restringir el acceso a la página Temas

Cada tema será propiedad de un usuario, por lo que sólo los usuarios registrados podrán solicitar la página de temas. Añade el siguiente código a *learning\_logs/views.py*:

**learning\_logs/views.py**

from django.shortcuts import render, redirect  
from django.contrib.auth.decorators import login\_required  
  
from .models import Topic, Entry  
--snip--  
  
@login\_required  
def topics(request):  
 """Show all topics."""  
 --snip--

Primero importamos la función login\_required(). Aplicamos login\_required() como decorador a la función de vista topics() anteponiendo a login\_required el símbolo @. Como resultado, Python sabe que debe ejecutar el código de login\_required() antes que el de topics().

El código en login\_required() comprueba si un usuario está conectado, y Django ejecuta el código en topics() sólo si lo está. Si el usuario no ha iniciado sesión, se le redirige a la página de inicio de sesión.

Para que esta redirección funcione, tenemos que modificar *settings.py* para que Django sepa dónde encontrar la página de inicio de sesión. Añade lo siguiente al final de *settings.py*:

**settings.py**

--snip--  
# My settings.  
LOGIN\_REDIRECT\_URL = 'learning\_logs:index'  
LOGOUT\_REDIRECT\_URL = 'learning\_logs:index'  
LOGIN\_URL = 'accounts:login'

Ahora, cuando un usuario no autenticado solicite una página protegida por el decorador @login\_required, Django enviará al usuario a la URL definida por LOGIN\_URL en *settings.py*.

Puedes probar esta configuración cerrando la sesión de cualquier cuenta de usuario y accediendo a la página de inicio. Haz clic en el enlace **Temas**, que debería redirigirte a la página de inicio de sesión. A continuación, inicia sesión en cualquiera de tus cuentas y, desde la página de inicio, vuelve a hacer clic en el enlace **Temas**. Deberías poder acceder a la página de temas.

#### Restringir el acceso a través del Registro de Aprendizaje

Django facilita la restricción de acceso a las páginas, pero tienes que decidir qué páginas proteger. Lo mejor es pensar primero qué páginas deben estar libres de restricciones, y luego restringir todas las demás páginas del proyecto. Puedes corregir fácilmente el acceso excesivamente restringido, y es menos peligroso que dejar páginas sensibles sin restringir.

En Learning Log, mantendremos sin restricciones la página de inicio y la página de registro. Restringiremos el acceso a todas las demás páginas.

Aquí tienes *learning\_logs/views.py* con los decoradores @login\_required aplicados a todas las vistas excepto a index():

**learning\_logs/views.py**

--snip--  
@login\_required  
def topics(request):  
 --snip--  
  
@login\_required  
def topic(request, topic\_id):  
 --snip--  
  
@login\_required  
def new\_topic(request):  
 --snip--  
  
@login\_required  
def new\_entry(request, topic\_id):  
 --snip--  
  
@login\_required  
def edit\_entry(request, entry\_id):  
 --snip--

Intenta acceder a cada una de estas páginas mientras estás desconectado; deberías ser redirigido a la página de inicio de sesión. Tampoco podrás hacer clic en enlaces a páginas como new\_topic. Pero si introduces la URL *http://localhost:8000/new\_topic/*, serás redirigido a la página de inicio de sesión. Deberías restringir el acceso a cualquier URL que sea de acceso público y esté relacionada con datos privados del usuario.

### Conexión de datos a determinados usuarios

A continuación, tenemos que conectar los datos al usuario que los envió. Sólo tenemos que conectar los datos más altos de la jerarquía a un usuario, y los datos de nivel inferior le seguirán. En Learning Log, los temas son el nivel más alto de datos de la aplicación, y todas las entradas están conectadas a un tema. Mientras cada tema pertenezca a un usuario concreto, podremos rastrear la propiedad de cada entrada en la base de datos.

Modificaremos el modelo Topic añadiendo una relación de clave foránea a un usuario. A continuación, tendremos que migrar la base de datos. Por último, modificaremos algunas de las vistas para que sólo muestren los datos asociados al usuario conectado en ese momento.

#### Modificar el modelo de temas

La modificación de *models.py* son sólo dos líneas:

**models.py**

from django.db import models  
from django.contrib.auth.models import User  
  
class Topic(models.Model):  
 """A topic the user is learning about."""  
 Text = models.CharField(max\_length=200)  
 date\_added = models.DateTimeField(auto\_now\_add=True)  
 owner = models.ForeignKey(User, on\_delete=models.CASCADE)  
  
 def \_\_str\_\_(self):  
 """Return a string representing the topic."""  
 Return self.text  
  
class Entry(models.Model):  
 --snip--

Importamos el modelo User de django.contrib.auth. A continuación, añadimos un campo owner a Topic, que establece una relación de clave foránea con el modelo User. Si se elimina un usuario, se eliminarán también todos los temas asociados a ese usuario.

#### Identificar a los usuarios existentes

Cuando migremos la base de datos, Django modificará la base de datos para que pueda almacenar una conexión entre cada tema y un usuario. Para realizar la migración, Django necesita saber qué usuario asociar a cada tema existente. Lo más sencillo es empezar asignando todos los temas existentes a un usuario, por ejemplo, el superusuario. Pero antes, necesitamos conocer el ID de ese usuario.

Veamos los ID de todos los usuarios creados hasta ahora. Inicia una sesión shell de Django y emite los siguientes comandos:

(ll\_env)learning\_log$ python manage.py shell  
❶ >>> from django.contrib.auth.models import User  
❷ >>> User.objects.all()  
<QuerySet [<User: ll\_admin>, <User: eric>, <User: willie>]>  
❸ >>> for user in User.objects.all():  
... print(user.username, user.id)  
...  
ll\_admin 1  
eric 2  
willie 3  
>>>

Primero importamos el modelo User a la sesión shell ❶. A continuación, miramos todos los usuarios que se han creado hasta el momento ❷. La salida muestra tres usuarios para mi versión del proyecto: ll\_admin, eric, y willie.

A continuación, recorremos la lista de usuarios e imprimimos el nombre de usuario y el ID ❸ de cada usuario . Cuando Django pregunte a qué usuario asociar los temas existentes, utilizaremos uno de estos valores de ID.

#### Migrar la base de datos

Ahora que conocemos los ID, podemos migrar la base de datos. Cuando lo hagamos, Python nos pedirá que conectemos el modelo Topic a un propietario concreto temporalmente o que añadamos un valor por defecto a nuestro archivo *models.py* para indicarle qué hacer. Elige la opción **1**:

❶ (ll\_env)learning\_log$ python manage.py makemigrations learning\_logs  
❷ It is impossible to add a non-nullable field 'owner' to topic without  
specifying a default. This is because...  
❸ Please select a fix:  
 1) Provide a one-off default now (will be set on all existing rows with a  
 null value for this column)  
 2) Quit and manually define a default value in models.py.  
❹ Select an option: 1  
❺ Please enter the default value now, as valid Python  
The datetime and django.utils.timezone modules are available...  
Type 'exit' to exit this prompt  
❻ >>> 1  
Migrations for 'learning\_logs':  
 learning\_logs/migrations/0003\_topic\_owner.py  
- Add field owner to topic  
(ll\_env)learning\_log$

Comenzamos emitiendo el comando makemigrations ❶. En la salida, Django indica que estamos intentando añadir un campo obligatorio (*non-nullable*) a un modelo existente (topic) sin valor por defecto especificado ❷. Django nos da dos opciones: podemos proporcionar un valor por defecto ahora mismo, o podemos abandonar y añadir un valor por defecto en *models.py* ❸. Aquí he elegido la primera opción ❹. A continuación, Django nos pide que introduzcamos el valor por defecto ❺.

Para asociar todos los temas existentes al usuario administrador original, ll\_admin, he introducido el ID de usuario de 1 ❻. Puedes utilizar el ID de cualquier usuario que hayas creado; no tiene por qué ser un superusuario. A continuación, Django migra la base de datos utilizando este valor y genera el archivo de migración *0003\_topic\_owner.py*, que añade el campo owner al modelo Topic.

Ahora podemos ejecutar la migración. Introduce lo siguiente en un entorno virtual activo:

(ll\_env)learning\_log$ python manage.py migrate  
Operations to perform:  
 Apply all migrations: admin, auth, contenttypes, learning\_logs, sessions  
Running migrations:  
❶ Applying learning\_logs.0003\_topic\_owner... OK  
(ll\_env)learning\_log$

Django aplica la nueva migración, y el resultado es OK ❶.

Podemos verificar que la migración ha funcionado como se esperaba en una sesión de shell, de la siguiente forma:

>>> from learning\_logs.models import Topic  
>>> for topic in Topic.objects.all():  
... print(topic, topic.owner)  
...  
Chess ll\_admin  
Rock Climbing ll\_admin  
>>>

Importamos Topic de learning\_logs.models y luego recorremos en bucle todos los temas existentes, imprimiendo cada tema y el usuario al que pertenece. Puedes ver que cada tema pertenece ahora al usuario ll\_admin. (Si obtienes un error al ejecutar este código, intenta salir de la shell e iniciar una nueva shell).

## Nota

Puedes simplemente reiniciar la base de datos en lugar de migrar, pero así perderás todos los datos existentes. Es una buena práctica aprender a migrar una base de datos manteniendo la integridad de los datos de los usuarios. Si quieres empezar con una base de datos nueva, ejecuta el comando python manage.py flush para reconstruir la estructura de la base de datos. Tendrás que crear un nuevo superusuario, y todos tus datos habrán desaparecido.

### Restringir el acceso a los temas a los usuarios adecuados

Actualmente, si estás conectado, podrás ver todos los temas, independientemente del usuario con el que estés conectado. Cambiaremos eso mostrando a los usuarios sólo los temas que les pertenecen.

Haz el siguiente cambio en la función topics() en *views.py*:

**learning\_logs/views.py**

--snip--  
@login\_required  
def topics(request):  
 """Show all topics."""  
 topics = Topic.objects.filter(owner=request.user).order\_by('date\_added')  
 context = {'topics': topics}  
 return render(request, 'learning\_logs/topics.html', context)  
--snip--

Cuando un usuario inicia sesión, el objeto request tiene un atributo request.user establecido, que contiene información sobre el usuario. La consulta Topic.objects.filter(owner=request.user) indica a Django que recupere sólo los objetos Topic de la base de datos cuyo atributo owner coincida con el usuario actual. Como no vamos a cambiar la forma en que se muestran los temas, no necesitamos cambiar en absoluto la plantilla de la página de temas.

Para ver si esto funciona, inicia sesión como el usuario al que conectaste todos los temas existentes, y ve a la página de temas. Deberías ver todos los temas. Ahora cierra la sesión y vuelve a entrar como un usuario diferente. Deberías ver el mensaje "Aún no se ha añadido ningún tema".

### Proteger los temas de un usuario

Aún no hemos restringido el acceso a las páginas de temas, por lo que cualquier usuario registrado podría probar un montón de URLs (como *http://localhost:8000/topics/1/*) y recuperar las páginas de temas que coincidan.

Pruébalo tú mismo. Mientras estás conectado como el usuario propietario de todos los temas, copia la URL o anota el ID en la URL de un tema, y luego desconéctate y vuelve a conectarte como un usuario diferente. Introduce la URL de ese tema. Deberías poder leer las entradas, aunque hayas iniciado sesión como un usuario diferente.

Arreglaremos esto ahora realizando una comprobación antes de recuperar las entradas solicitadas en la función de vista topic():

**learning\_logs/views.py**

from django.shortcuts import render, redirect  
from django.contrib.auth.decorators import login\_required  
❶ from django.http import Http404  
  
--snip--  
@login\_required  
def topic(request, topic\_id):  
 """Show a single topic and all its entries."""  
 topic = Topic.objects.get(id=topic\_id)  
 # Make sure the topic belongs to the current user.  
❷ if topic.owner != request.user:  
 raise Http404  
  
 entries = topic.entry\_set.order\_by('-date\_added')  
 context = {'topic': topic, 'entries': entries}  
 return render(request, 'learning\_logs/topic.html', context)  
--snip--

Una respuesta 404 es una respuesta de error estándar que se devuelve cuando un recurso solicitado no existe en un servidor. Aquí importamos la excepción Http404 ❶, que lanzaremos si el usuario solicita un tema al que no debería tener acceso. Tras recibir una solicitud de tema, nos aseguramos de que el usuario del tema coincide con el usuario conectado en ese momento antes de mostrar la página. Si el propietario del tema solicitado no coincide con el usuario actual, lanzamos la excepción Http404 ❷, y Django devuelve una página de error 404.

Ahora, si intentas ver las entradas de los temas de otro usuario, verás un mensaje de "Página no encontrada" de Django. En el Capítulo 20, configuraremos el proyecto para que los usuarios vean una página de error adecuada en lugar de una página de depuración.

### Proteger la página edit\_entry

Las páginas edit\_entry tienen URLs de la forma *http://localhost:8000/edit\_entry/entry\_id/*, donde entry\_id es un número. Protejamos esta página para que nadie pueda utilizar la URL para acceder a las entradas de otra persona:

**learning\_logs/views.py**

--snip--  
@login\_required  
def edit\_entry(request, entry\_id):  
 """Edit an existing entry."""  
 entry = Entry.objects.get(id=entry\_id)  
 topic = entry.topic  
 if topic.owner != request.user:  
 raise Http404  
  
 if request.method != 'POST':  
 --snip--

Recuperamos la entrada y el tema asociado a esta entrada. A continuación, comprobamos si el propietario del tema coincide con el usuario conectado en ese momento; si no coinciden, lanzamos una excepción Http404.

### Asociar nuevos temas al usuario actual

Actualmente, la página para añadir nuevos temas está rota porque no asocia los nuevos temas a ningún usuario en particular. Si intentas añadir un nuevo tema, verás el mensaje IntegrityError junto con NOT NULL constraint failed: learning\_logs\_topic.owner\_id. Django está diciendo que no puedes crear un nuevo tema sin especificar un valor para el campo owner del tema.

Hay una solución sencilla para este problema, porque tenemos acceso al usuario actual a través del objeto request. Añade el siguiente código, que asocia el nuevo tema con el usuario actual:

**learning\_logs/views.py**

--snip--  
@login\_required  
def new\_topic(request):  
 --snip--  
 else:  
 # POST data submitted; process data.  
 form = TopicForm(data=request.POST)  
 if form.is\_valid():  
❶ new\_topic = form.save(commit=False)  
❷ new\_topic.owner = request.user  
❸ new\_topic.save()  
 return redirect('learning\_logs:topics')  
  
 # Display a blank or invalid form.  
 context = {'form': form}  
 return render(request, 'learning\_logs/new\_topic.html', context)  
 --snip--

Cuando llamamos por primera vez a form.save(), pasamos el argumento commit=False porque necesitamos modificar el nuevo tema antes de guardarlo en la base de datos ❶. A continuación, establecemos el atributo owner del nuevo tema con el usuario actual ❷. Por último, llamamos a save() en la instancia del tema que acabamos de definir ❸. Ahora el tema tiene todos los datos necesarios y se guardará correctamente.

Deberías poder añadir tantos temas nuevos como quieras para tantos usuarios diferentes como desees. Cada usuario sólo tendrá acceso a sus propios datos, tanto si está viendo datos, introduciendo datos nuevos o modificando datos antiguos.

## Pruébalo tú mismo

19-3. Refactorización: Hay dos lugares en *views.py* en los que nos aseguramos de que el usuario asociado a un tema coincide con el usuario conectado en ese momento. Pon el código para esta comprobación en una función llamada check\_topic\_owner(), y llama a esta función cuando sea apropiado.

19-4. Proteger nueva\_entrada: Actualmente, un usuario puede añadir una nueva entrada al registro de aprendizaje de otro usuario introduciendo una URL con el ID de un tema perteneciente a otro usuario. Evita este ataque comprobando que el usuario actual es el propietario del tema de la entrada antes de guardar la nueva entrada.

19-5. Blog protegido: En tu proyecto de Blog, asegúrate de que cada entrada del blog está conectada a un usuario concreto. Asegúrate de que todas las entradas son accesibles públicamente, pero sólo los usuarios registrados pueden añadir entradas y editar las existentes. En la vista que permite a los usuarios editar sus entradas, asegúrate de que el usuario está editando su propia entrada antes de procesar el formulario.

## Resumen

En este capítulo, has aprendido cómo los formularios permiten a los usuarios añadir nuevos temas y entradas, y editar las entradas existentes. Luego aprendiste a implementar cuentas de usuario. Proporcionaste a los usuarios existentes la capacidad de iniciar y cerrar sesión, y utilizaste la dirección UserCreationForm predeterminada de Django para permitir a la gente crear nuevas cuentas.

Después de construir un sistema sencillo de autenticación y registro de usuarios, restringiste el acceso a los usuarios registrados para determinadas páginas utilizando el decorador @login\_required. A continuación, asignaste datos a usuarios específicos mediante una relación de clave foránea. También aprendiste a migrar la base de datos cuando la migración requiere que especifiques algunos datos por defecto.

Por último, aprendiste a asegurarte de que un usuario sólo puede ver los datos que le pertenecen modificando las funciones de vista. Recuperaste los datos apropiados utilizando el método filter(), y comparaste el propietario de los datos solicitados con el usuario actualmente conectado.

Puede que no siempre sea inmediatamente obvio qué datos debes poner a disposición y qué datos debes proteger, pero esta habilidad se adquiere con la práctica. Las decisiones que hemos tomado en este capítulo para proteger los datos de nuestros usuarios también ilustran por qué trabajar con otros es una buena idea cuando se construye un proyecto: que otra persona revise tu proyecto hace más probable que detectes las áreas vulnerables.

Ahora tienes un proyecto totalmente operativo en tu máquina local. En el último capítulo, darás estilo al Registro de Aprendizaje para hacerlo visualmente atractivo, y desplegarás el proyecto en un servidor para que cualquier persona con acceso a Internet pueda registrarse y crear una cuenta.