Práctica 6: Desarrollo de una aplicación Web con Flask (IV)

Ejercicio 1
Publicación de mensajes de respuesta

Hasta ahora, tu aplicación no permite publicar respuestas a mensajes. En este ejercicio vamos a implementar esta funcionalidad en la vista de mensajes. Para simplificar, la aplicación no permitirá responder a mensajes que sean, a su vez, respuestas a otros mensajes. Es decir, no se permitirá anidar respuestas.

En primer lugar, modifica la plantilla de la vista de mensajes añadiendo un nuevo formulario que permita responder al mensaje. Este formulario será casi igual que el de publicación de mensajes nuevos en la vista principal, y también se enviará al mismo controlador, con la diferencia de que incluirá un control adicional de tipo **`hidden`** que especifique el identificador del mensaje al que se está respondiendo. Los controles ocultos no se muestran en el navegador, pero sus datos se envían junto con los datos de los demás controles del formulario cuando el usuario lo envía:

```
<input type="hidden" name="response_to" value="{{ post.id }}">
```

Ahora, el controlador que crea un mensaje nuevo, el que recibe este formulario, necesita leer el parámetro **`response_to`** para saber si se trata de un mensaje de respuesta:

- Si no es un mensaje de respuesta, ese parámetro no estará presente, y la llamada **`request.form.get("response_to")`** devolverá **`None`**.
- Si es un mensaje de respuesta, ese parámetro estará presente, y la llamada **`request.form.get("response_to")`** devolverá una cadena con el identificador del mensaje al que se está respondiendo. En ese caso, lee ese objeto mensaje de la base de datos (si no existe, aborta con un error 404).

Cuando crees el objeto mensaje en el controlador, inicializa también su atributo **`response_to`**, con un valor **`None`** en el primer caso o con el objeto del mensaje al que se está respondiendo en el segundo caso.

Por último, modifica la redirección al final del controlador:

- Si el mensaje que has creado no es una respuesta, simplemente redirige a la vista de ese mensaje (tu código ya está haciendo eso).
- Si el mensaje que has creado es una respuesta, redirige a la vista del mensaje original al que se está respondiendo. En el siguiente ejercicio modificarás esa vista para que muestre el mensaje original junto con las respuestas a él.

Ejercicio 2
Mostrar mensajes de respuesta

Hasta ahora, la vista de mensajes muestra un mensaje sin sus respuestas. Modifícala para que muestre las respuestas, si las hay, ordenadas de más reciente a menos reciente.

Para obtener la lista ordenada de respuestas, busca en la base de datos, desde el controlador, todas las respuestas al mensaje y ordénalas:

```
responses = (
    model.Message.query.filter_by(response_to=message)
    .order_by(model.Message.timestamp.desc())
    .all()
)
```

Ejercicio 3
Ocultar los mensajes de respuesta en algunas vistas

Dado que los mensajes de respuesta son ciudadanos de segunda clase en nuestra aplicación, no queremos que se muestren en la vista principal ni en la vista de usuario. Además, no queremos mostrar la vista de mensajes para los mensajes de respuesta.

Aplica los siguientes cambios:

1. En la vista principal, obtén los 10 mensajes más recientes *que no sean respuestas*:

   ```
   model.Message.query.filter_by(response_to=None)
       .order_by(model.Message.timestamp.desc())
       .limit(10)
       .all()
   ```

2. En la vista de perfil de usuario, obtén los mensajes del usuario que no sean respuestas. Aplica el siguiente filtro a la consulta: **`filter_by(user=user,`**

`response_to=None)`, que representa un *AND lógico* de ambas condiciones.
3. En la vista de mensajes, aborta con un error 403 (*Forbidden*) si el mensaje a mostrar es una respuesta.
4. En la plantilla para mensajes individuales (**`post_template.html`**) no se debe mostrar el enlace a la vista de mensajes si el mensaje es una respuesta. En una plantilla de Jinja, puedes comprobar si un mensaje es una respuesta con el siguiente código:

```
{% if post.response_to is none %}
    (...)
{% endif %}
```

Ejercicio 4
Seguir a usuarios

Implementemos ahora una nueva funcionalidad para que los usuarios puedan seguir a otros usuarios de la aplicación.

El primer paso consiste en modificar el modelo de datos. Como un usuario puede seguir a muchos usuarios y ser, a su vez, seguido por muchos usuarios, existe una relación **`muchos a muchos`** de la tabla de usuarios consigo misma. Las relaciones muchos a muchos se modelan en SQLAlchemy declarando una tabla intermedia para la relación (lo mismo que harías en SQL). Esa relación se puede declarar al principio de **`model.py`**, antes de las otras dos clases, de la siguiente manera:

```
class FollowingAssociation(db.Model):
    follower_id = db.Column(
        db.Integer, db.ForeignKey("user.id"), primary_key=True, nullable=False
    )
    followed_id = db.Column(
        db.Integer, db.ForeignKey("user.id"), primary_key=True, nullable=False
    )
```

Como puedes ver, la tabla consiste en dos columnas: **`follower_id`** y **`followed_id`**. Ambas son claves ajenas que hacen referencia a filas de la tabla de usuarios. Además, la combinación de ambas columnas forma la clave primaria de la tabla.

La relación en sí se declara en la clase **`User`**. Es un poco más complicada de lo normal por ser una relación de una tabla consigo misma:

```
class User(flask_login.UserMixin, db.Model):
    (...)
    following = db.relationship(
        "User",
        secondary=FollowingAssociation.__table__,
        primaryjoin=FollowingAssociation.follower_id == id,
        secondaryjoin=FollowingAssociation.followed_id == id,
        backref="followers",
    )
```

La declaración de la relación **`following`**, que lista los usuarios a los que sigue el usuario representado por el objeto, indica lo siguiente:

- La relación va desde la tabla actual (**`User`**) hasta la tabla nombrada en el primer parámetro de la llamada a **`db.relationship`** (de nuevo, **`User`**).
- La relación pasa por la tabla intermedia, la correspondiente a la clase **`FollowingAssociation`** que declaramos antes.
- Dado que hay dos claves ajenas que van desde **`FollowingAssociation`** hasta **`User`**, tenemos que explicitar cómo se hace la unión de la tabla **`User`** con la tabla **`FollowingAssociation`** y de vuelta a la otra vista de la tabla **`User`** para construir la lista **`following`**. Los parámetros **`primaryjoin`** y **`secondaryjoin`** especifican que la lista **`following`** se construye uniendo el usuario actual como seguidor y el usuario al otro extremo de la relación como el seguido.
- La lista inversa **`followers`** se crea mediante el parámetro **`backref`**. Contendrá la lista de seguidores del usuario actual.

Finalmente, debes dejar que SQLAlchemy actualice tu base de datos. Desde una terminal de línea de comandos activa tu entorno virtual y ejecuta los mismos comandos que ejecutaste en un laboratorio anterior para crear las tablas en sí:

```
from microblog import db, create_app
db.create_all(app=create_app())
```

El modelo ya está completo. Los objetos **`User`** contendrán dos nuevos atributos: **`following`** y **`followers`**. Ambos serán listas de Python, y por tanto podrás operar con ellas como lo harías con listas normales. En el siguiente ejercicio aprenderás a usar esos atributos para hacer que los usuarios sigan o dejen de seguir a otros usuarios.

Ejercicio 5
Controladores para seguir y dejar de seguir

Ahora que el modelo de base de datos soporta la relación seguidor / seguido entre usuarios, vamos a permitir a los usuarios, cuando visiten el perfil público de otro usuario, seguirlo o dejar de seguirlo.

En primer lugar, programa un nuevo controlador en el *blueprint* **`main`**. Este controlador hará que el usuario actual (el que tiene la sesión iniciada) siga a otro usuario. Este último se recibirá como parámetro en la URL:

```
@bp.route("/follow/<int:user_id>", methods=["POST"])
@flask_login.login_required
def follow(user_id):
    (...)
```

Este controlador necesita obtener el otro usuario de la base de datos (y devolver un error 404 Not Found si no está ahí) y asegurarse de que ambos usuarios no sean el mismo y de que el usuario actual no esté siguiendo ya al otro usuario. En estas dos últimas situaciones también se debe devolver un error (por ejemplo, 403 Forbidden):

```
user = model.User.query.filter_by(id=user_id).first_or_404()
if flask_login.current_user in user.followers:
    abort(403, "Already following that user")
elif flask_login.current_user == user:
    abort(403, "Users cannot follow themselves")
```

Fíjate em cómo la expresión **`flask_login.current_user in user.followers`** se ha usado para comprobar si el usuario actual ya sigue al otro usuario. También podríamos haberlo comprobado al revés, con exactamente el mismo resultado: **`user in flask_login.current_user.following`**.

A continuación, el controlador tiene que hacer que el usuario actual siga al otro. Para ello, basta con añadir el usuario actual a la lista **`followers`** del usuario a seguir (**`user.followers.append(flask_login.current_user)`**) o, al revés, añadir el usuario a seguir a la lista **`following`** del usuario actual. Solo necesitas una de las dos instrucciones anteriores, ya que la otra será implícita. Después, guarda los cambios con **`db.session.commit()`** y redirige al cliente al perfil público del usuario a seguir.

Programa otro controlador para *dejar de seguir* a un usuario. Será bastante similar al que acabas de programar. Para dejar de seguir a un usuario, basta con llamar a:

```
user.followers.remove(flask_login.current_user)
```

No puedes probar estos dos nuevos controladores todavía. Para ello, necesitarías añadir un formulario para seguir o dejar de seguir a un usuario en la vista del perfil de usuario. Eso lo harás en el siguiente ejercicio.

Ejercicio 6
Formularios para seguir y dejar de seguir a usuarios

Primero, modifica el controlador de la vista del perfil de usuario para que envíe un nuevo parámetro llamado, por ejemplo, **`follow_button`** a la plantilla de la vista. Será una cadena de texto que tomará los valores "none" (cuando el usuario actual y el usuario mostrado son el mismo), "follow" (cuando el usuario actual aún no está siguiendo al usuario mostrado en esa vista de perfil) y "unfollow" (cuando el usuario actual ya está siguiendo al usuario mostrado en esa vista de perfil).

Con dicho parámetro, la vista sabrá ahora qué formulario mostrar: un formulario para seguir, un formulario para dejar de seguir o ningún formulario. Esos formularios solo necesitan un control, que será un botón de envío con el texto apropiado:

```
{% if follow_button == "follow" %}
    <form action="{{ url_for('main.follow', user_id=user.id) }}" method="post">
        <input type="submit" value="Follow">
    </form>
{% elif follow_button == "unfollow" %}
    <form action="{{ url_for('main.unfollow', user_id=user.id) }}" method="post">
        <input type="submit" value="Unfollow">
    </form>
{% endif %}
```

Fíjate en cómo se construye la URL del controlador de destino con **`url_for`**, incluyendo el id del usuario a seguir o dejar de seguir. Fíjate también en que el formulario se enviará con una petición **`POST`**, ya que es lo apropiado para una acción como seguir o dejar de seguir.

Ahora ya puedes probar la nueva funcionalidad de seguir y dejar de seguir.

Ejercicio 7
Mostrar la lista de seguidores y seguidos en la vista de perfil de usuario

Actualiza la vista de perfil de usuario para que muestre la lista de usuarios que siguen al usuario mostrado y la lista de usuarios a los que sigue el usuario mostrado. Cada nombre de usuario debe enlazar a su perfil público. Necesitarás hacer, en la plantilla, un bucle sobre el atributo **`following`** del usuario y otro sobre su atributo **`followers`**.

Ejercicio 8
Mostrar solo mensajes de los usuarios a los que se sigue en la vista principal

Para completar la funcionalidad de seguir y dejar de seguir, necesitamos modificar la consulta que hace el controlador de la vista principal para que obtenga, en lugar de los 10 mensajes más recientes de cualquier usuario, los 10 mensajes más recientes de los usuarios a los que sigue el usuario actual. La nueva consulta es:

```
following = db.aliased(model.User)
messages = (
    model.Message.query.join(model.User)
    .join(following, model.User.followers)
    .filter(following.id == flask_login.current_user.id, model.Message.response_to == None)
    .order_by(model.Message.timestamp.desc())
    .limit(10)
    .all()
)
```

La consulta combina la tabla de mensajes con la tabla de usuarios y luego otra vez con la tabla de usuarios. Asegúrate de entenderla:

- Dado que necesitamos unir la tabla de usuarios consigo misma, tenemos que darle un alias a una *vista* de la tabla. Su alias será **`following`** y actuará como si fuera una tabla real, igual que sucede con los alias de tablas en el lenguaje SQL.
- La consulta devuelve mensajes porque la hemos construido sobre **`model.Message.query`**.
- La sentencia **`join`** entre mensajes y usuarios es sencilla, dado que solo hay una relación entre ellos.
- La sentencia **`join`** entre usuarios y su vista (**`following`**) necesita una declaración extra de la relación a usar, ya que hay dos relaciones posibles entre ellos. Especificamos que la vista de la derecha (la que tiene un alias) contendrá los seguidores (**`model.User.followers`**). Por tanto, la de la izquierda contendrá los usuarios seguidos.
- Filtramos las filas donde el usuario actual esté siguiendo a cualquier otro usuario.
- También filtramos las filas donde el mensaje no sea una respuesta a otro mensaje. Solo queremos mostrar mensajes originales aquí.
- Finalmente, ordenamos los mensajes por fecha de publicación, de más reciente a menos reciente, y limitamos el resultado a 10 mensajes.

Comprueba que la vista funcione y muestre los mensajes correctos.

CC BY NC SA