Práctica 4: Desarrollo de una aplicación Web con Flask (II)

Ejercicio 1
Declaración de la conexión con la base de datos

En lugar de emitir directamente comandos SQL desde el código, utilizaremos un mapeador objeto-relacional (ORM), que almacenará y recuperará automáticamente los objetos Python (usuarios, mensajes, etc.) en la base de datos. En concreto, utilizaremos SQLAlchemy, una conocida biblioteca del ecosistema Python. Aunque la utilizaremos aquí para conectarnos a servidores MySQL o MariaDB, es compatible con cualquiera de los principales sistemas de gestión de bases de datos basados en SQL, como por ejemplo el de Oracle.

El paquete **`Flask-SQLAlchemy`** integra SQLAlchemy en aplicaciones Flask. Lo primero que haremos será instalar el módulo. Activa el entorno virtual que creaste en la práctica anterior e instala los paquetes **`Flask-SQLAlchemy`** y **`mysqlclient`**:

```
pip install flask-sqlalchemy==2.5.1 mysqlclient
```

El primer paso será cargar y configurar el módulo **`Flask-SQLAlchemy`**. Reemplaza el contenido de **`__init__.py`** con el siguiente código, el cual crea una base de datos SQLite en el fichero **`microblog.db`** en el directorio de la aplicación:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()


def create_app(test_config=None):
    app = Flask(__name__)
    app.config["SECRET_KEY"] = b"\x8c\xa5\x04\xb3\x8f\xa1<\xef\x9bY\xca/*\xff\x12\xfb"
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///microblog.db"
    app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
    db.init_app(app)

    # Register blueprints:
    from . import main

    app.register_blueprint(main.bp)
    return app
```

Fíjate en algunas cosas del código:

- Estamos diciéndole a SQLAlchemy cómo conectarse a la base de datos con el parámetro de configuración **`SQLALCHEMY_DATABASE_URI`**. En concreto, estamos definiendo el acceso para el usuario **`microblog`** con contraseña **`waDBlog`** al servidor en el nombre de dominio **`localhost`** y base de datos con el nombre **`Microblog`**.
- Usaremos el objeto **`db`** para acceder a la base de datos a través de SQLAlchemy cuando sea necesario.
- Estamos importando el módulo **`main`** desde dentro de la función **`create_app`**. El motivo de este cambio es evitar un error de importación circular en el siguiente ejercicio.

Como todavía no podemos probar este código, pasa al siguiente ejercicio directamente.

Ejercicio 2
Definición del esquema de datos

Ahora podemos ya definir el esquema de datos. Reemplaza el contenido del módulo **`model.py`** con esta nueva versión:

```
from . import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(128), unique=True, nullable=False)
    name = db.Column(db.String(64), nullable=False)
    password = db.Column(db.String(100), nullable=False)
    messages = db.relationship('Message', backref='user', lazy=True)

class Message(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    text = db.Column(db.String(512), nullable=False)
    timestamp = db.Column(db.DateTime(), nullable=False)
    response_to_id = db.Column(db.Integer, db.ForeignKey('message.id'))
    response_to = db.relationship(
        'Message',
        backref='responses',
        remote_side=[id],
        lazy=True
    )
```

`User` y **`Message`**, además de ser clases normales de Python, ahora definen también el esquema de la base de datos, porque extienden la clase **`Model`** de SQLAlchemy.

La clase **`User`** define una tabla de base de datos, que por defecto se llamará "user", con las siguientes características:

- Columna **`id`**: clave primaria de la tabla.

- Columna **`email`**: cadena de texto, que debe ser única y contener como mucho 128 caracteres.
- Columna **`name`**: cadena de texto, con 64 caracteres como mucho.
- Columna **`password`**: cadena de texto, con 100 caracteres como mucho.
- Una relación uno a muchos con la tabla que almacena los mensajes (un usuario puede ser autor de muchos mensajes). El atributo **`messages`** de los objetos de la clase **`User`** contendrá la lista de todos los mensajes (objetos de la clase **`Message`**) de este usuario. La relación se marca como **`lazy`**, lo que significa que, por motivos de eficiencia, los mensajes de un usuario no se cargarán de la base de datos hasta que se acceda a ellos. En su lugar, se cargarán en memoria la primera vez que tu código acceda al atributo **`messages`**. Además, esta declaración añadirá un atributo **`user`** a la clase **`Message`**, de tal forma que se pueda recuperar automáticamente el objeto del creador de cada objeto de dicha clase.

La clase **`Message`** define una tabla de base de datos, que por defecto se llamará "message", con las siguientes características:

- Columna **`id`**: clave primaria de la tabla.
- Columna **`user_id`**: autor del mensaje, clave ajena a la tabla de usuarios. Es el otro extremo de la relación definida arriba.
- Columna **`text`**: cadena de texto con el contenido del mensaje.
- Columna **`timestamp`**: fecha y hora de creación del mensaje.
- Columna **`response_to_id`**: clave ajena a la tabla de mensajes. Define a qué mensaje responde este mensaje, si es que responde a alguno. Si no responde a ningún mensaje, se establecerá el valor **`null`**.
- Una relación muchos a uno con la misma tabla (muchos mensajes pueden responder a un mensaje). Define un atributo **`response_to`** para los objetos de la clase **`Message`**, que proporciona el objeto **`Message`** al que responde este mensaje, o **`null`** en caso contrario, así como el un atributo **`responses`** que almacena la lista de mensajes que responden a este mensaje.

El modo de crear objetos de tipo **`User`** y **`Message`** cambia ahora un poco. Ajusta tu código en **`main.py`** siguiendo el ejemplo que aparece a continuación:

```
user = model.User(email="mary@example.com", name="mary")
posts = [
    model.Message(
        user=user,
        text="Test post",
        timestamp=datetime.datetime.now(dateutil.tz.tzlocal()),
    ),
    model.Message(
        user=user,
        text="Another post",
        timestamp=datetime.datetime.now(dateutil.tz.tzlocal()),
    ),
]
```

Finalmente, para que SQLAlchemy cree automáticamente las tablas en tu base de datos según el esquema que acabas de definir, ejecuta un intérprete de Python: escribe **`python`** en tu línea de comandos (con tu entorno virtual activado). Dentro del terminal de Python, ejecuta las siguientes instrucciones:

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

Abre de nuevo un terminal de Python y crea algunos datos de ejemplo:

```
from microblog import db, create_app, model
app = create_app()
app.app_context().push()
import datetime
import dateutil
user = model.User(email="mary@example.com", name="Mary", password="pwd")
message = model.Message(
    user=user,
    text="First message",
    timestamp=datetime.datetime.now(dateutil.tz.tzlocal())
)
message2 = model.Message(
    user=user,
    text="Response message",
    timestamp=datetime.datetime.now(dateutil.tz.tzlocal()),
    response_to=message
)
message3 = model.Message(
    user=user,
    text="Response message 2",
    timestamp=datetime.datetime.now(dateutil.tz.tzlocal()),
    response_to=message
)
db.session.add(message2)
db.session.add(message3)
db.session.commit()
```

Las primeras líneas son solo el código necesario para arrancar una aplicación Flask. La función **`db.session.add`** marca los objetos que deben ser añadidos a la base de datos (observa que no es necesario añadir explícitamente **`user`** y **`message`** porque se añadirán automáticamente al contener **`message2`** y **`message3`** referencias a ellos). Por último, **`db.session.commit()`** guarda los objetos en la base de datos.

Comprueba que se han creado las filas correspondientes en la base de datos abriendo un nuevo intérprete de Python y ejecutando las siguientes instrucciones:

```
from microblog import db, create_app, model
app = create_app()
app.app_context().push()
messages = model.Message.query.all()
messages[0].text
messages[1].text
messages[1].response_to.text
messages[0].user.email
len(messages[0].responses)
messages[0].responses[0].text
```

Como puedes ver, acabas de consultar la base de datos sin usar sentencias SQL, y puedes acceder a los resultados de la consulta como instancias de las clases que has

definido.

Ejercicio 3
Registro de nuevos usuarios

Aunque autenticación y autorización parecen a primera vista lo mismo, no lo son. La autenticación consiste en identificar a los usuarios a partir de las credenciales que proporcionan. Estas credenciales suelen consistir en una dirección de correo electrónico y una contraseña, aunque hoy en día es buena práctica incluir también algún otro factor de autenticación.

La autorización consiste en, dada una recurso y un usuario que se ha autenticado previamente, verificar si ese usuario tiene permiso para acceder al recurso. Por ejemplo, para acceder al recurso que borra un mensaje concreto en una red social, solo deberían estar autorizados el usuario que creó el mensaje y ciertos administradores de la red social.

El siguiente paso para construir la aplicación es gestionar la autenticación de los usuarios, de forma que puedan registrarse, iniciar sesión y cerrar sesión. En este ejercicio empezaremos a trabajar en esa dirección permitiendo a los usuarios crear nuevas cuentas.

Primero, crea una nueva plantilla para el formulario de registro y guárdala como **`templates/auth/signup.html`**. Puedes usar el siguiente código de ejemplo o escribir el tuyo propio:

```
{% extends 'base.html' %}

{% block content %}
    <h2>Sign Up!!!</h2>
    <form action="{{ url_for('auth.signup') }}" method="post">
        <div>
            <label>Email: <input type="email" name="email" required/></label>
        </div>
        <div>
            <label>User name: <input type="text" name="username" required/></label>
        </div>
        <div>
            <label>Password: <input type="password" name="password" required/></label>
        </div>
        <div>
            <label>
                Repeat password:
                <input type="password" name="password_repeat" required/>
            </label>
        </div>
        <div>
            <input type="submit" value="Sign Up"/>
        </div>
    </form>
{% endblock %}
```

El formulario recogerá cuatro parámetros del usuario, llamados "email", "username", "password" y "password_repeat". Cuando el usuario haga clic en el botón *Sign Up*, se enviará una petición HTTP POST con esos parámetros a la ruta del controlador **`auth.signup_post`**, que programaremos a continuación.

Para mantener nuestro código modular, crearemos un nuevo *blueprint* para el código relacionado con la autenticación de usuarios. Guarda el fichero **`microblog/auth.py`** con el siguiente código:

```
from flask import Blueprint, render_template, request, redirect, url_for, flash
from . import db, bcrypt

from . import model

bp = Blueprint("auth", __name__)


@bp.route("/signup")
def signup():
    return render_template("auth/signup.html")


@bp.route("/signup", methods=["POST"])
def signup_post():
    email = request.form.get("email")
    username = request.form.get("username")
    password = request.form.get("password")
    # Check that passwords are equal
    if password != request.form.get("password_repeat"):
        return redirect(url_for("auth.signup"))
    # Check if the email is already at the database
    user = model.User.query.filter_by(email=email).first()
    if user:
        return redirect(url_for("auth.signup"))
    password_hash = bcrypt.generate_password_hash(password).decode("utf-8")
    new_user = model.User(email=email, name=username, password=password_hash)
    db.session.add(new_user)
    db.session.commit()
    return redirect(url_for("main.index"))
```

Estás definiendo dos controladores para la ruta **`/signup`**. El primero responderá a peticiones GET para esa ruta, ya que este es el método por defecto cuando no especificas uno. Este controlador simplemente mostrará el formulario de registro. El segundo responderá a peticiones POST. Recogerá los parámetros enviados desde el formulario de registro y creará el usuario en la base de datos si todo es correcto.

En el segundo controlador puedes ver cómo acceder a los parámetros de la petición que provienen del formulario, llamando al método **`request.form.get`** con el nombre del parámetro como argumento. Ten en cuenta que los nombres de los parámetros son los que has puesto como valor del atributo **`name`** de cada control del formulario.

Cuando todo va bien, se crea un nuevo objeto **`User`** y se almacena en la base de datos. A continuación, el usuario es redirigido (respuesta de redirección HTTP) a la página de inicio a través de la sentencia **`return redirect(...)`**. Si el email ya

existe en la base de datos o las contraseñas son diferentes, no se crea ningún usuario y el usuario es redirigido de nuevo al mismo formulario.

El módulo **`bcrypt`** se usa para encriptar las contraseñas en la base de datos con la función de *hash* de contraseñas **`bcrypt`**, de forma que estas sean difíciles de descifrar incluso si los atacantes consiguen una copia de la base de datos. Instálalo, en un terminal donde tu entorno virtual esté activado, con:

```
pip install flask-bcrypt
```

A continuación, cárgalo al principio del fichero **`__init__.py`**:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt

db = SQLAlchemy()
bcrypt = Bcrypt()
```

Después, necesitas registrar el *blueprint* en el módulo **`__init__.py`**. Haz lo mismo que ya estás haciendo para importar el *blueprint* principal (es decir, importa el módulo **`auth`** y regístralo).

Finalmente, ejecuta el servidor Flask y prueba tu código:

```
export FLASK_APP=microblog
flask --debug run
```

Prueba a rellenar el formulario de registro desde tu navegador. Prueba también a crear una cuenta para un email ya registrado y a crear una cuenta con campos de contraseña y repetición de contraseña que no coincidan.

Ejercicio 4
Realimentación en caso de errores de usuario

Cuando falla el registro de un usuario, no se envía ninguna realimentación al usuario, por lo que este no sabrá qué ha ido mal. Flask proporciona el mecanismo *flash* para pasar mensajes desde el contexto de una petición a la siguiente petición de la misma sesión de usuario. Puedes crear un mensaje *flash* en el controlador y luego mostrar dicho mensaje desde la plantilla del formulario cuando este se vuelva a mostrar.

Para usar el mecanismo *flash*, añade las siguientes tres llamadas al controlador que procesa los datos del formulario en **`auth.py`**:

```
(...)
# Check that passwords are equal
if password != request.form.get("password_repeat"):
    flash("Sorry, passwords are different")
    return redirect(url_for("auth.signup"))
# Check if the email is already at the database
user = model.User.query.filter_by(email=email).first()
if user:
    flash("Sorry, the email you provided is already registered")
    return redirect(url_for("auth.signup"))
(...)
flash("You've successfully signed up!")
return redirect(url_for("main.index"))
```

Para mostrar estos mensajes, añade el siguiente código en el lugar adecuado de la plantilla del formulario de registro:

```
{% with messages = get_flashed_messages() %}
    {% for message in messages %}
        <div class="notification">{{ message }}</div>
    {% endfor %}
{% endwith %}
```

Prueba esta nueva versión de la aplicación y comprueba que ahora ves la realimentación cuando se produce algún error en el registro de usuarios.

Ejercicio 5
Autenticación de usuarios (I)

Crea una plantilla para el formulario de inicio de sesión en **`templates/auth/login.html`**, así como un controlador en **`auth.py`** para la ruta **`/login`**, que simplemente le diga a Flask que represente el formulario. El formulario tiene que pedir un email y una contraseña, e incluir un enlace al formulario de registro para que los nuevos usuarios puedan crear una cuenta. Los datos tienen que enviarse al servidor a través de una petición **`POST`** a la ruta **`/login`**.

Modifica también el controlador de registro para que redirija hacia este formulario de inicio de sesión cuando un usuario se registre correctamente. De esta forma, los nuevos usuarios pueden iniciar sesión una vez se hayan registrado.

Crea un controlador en **`auth.py`** que reciba el email y la contraseña y los compruebe. A continuación se muestran los fragmentos de código relevantes para comprobar la autenticación:

```
# Complete the following two lines:
email = ...
password = ...
# Get the user with that email from the database:
user = model.User.query.filter_by(email=email).first()
if user and bcrypt.check_password_hash(user.password, password):
    # The user exists and the password is correct
    return redirect(url_for("main.index"))
```

```
else:
    # Wrong email and/or password
    # Complete this code to flash a message and redirect to the login form
    ...
```

Fíjate en que, como en la base de datos se guarda el valor de *hash* de las contraseñas en vez de las contraseñas en sí mismas, la comparación de contraseñas se tiene que hacer a través del módulo **`bcrypt`**. Internamente, este módulo calculará el valor de *hash* de la contraseña recibida del formulario y lo comparará con el que se guarda en el objeto de usuario que se ha recibido de la base de datos.

Ahora, modifica la plantilla del formulario de inicio de sesión para que muestre los mensajes *flash*, y comprueba el sistema de inicio de sesión con emails y contraseñas correctos e incorrectos.

Ejercicio 6
Autenticación de usuarios (II)

En el ejercicio anterior hemos creado el código para autenticar usuarios, pero todavía no estamos recordando quién ha iniciado sesión. En vez de programar esa parte manualmente, vamos a usar el paquete Flask-Login, que automatiza el seguimiento de sesiones para usuarios autenticados, así como el control de acceso a recursos que necesitan un usuario autenticado. Necesitas instalar este módulo desde un terminal:

```
pip install flask-login
```

Primero, activa el módulo **`Flask-Login`** en **`__init__.py`**. Importa el módulo y luego, en algún lugar dentro de la función **`create_app`**, antes de registrar los *blueprints*, inserta el siguiente código:

```
# With the other imports at the beginning:
from flask_login import LoginManager
(...)

# Inside create_app:
login_manager = LoginManager()
login_manager.login_view = 'auth.login'
login_manager.init_app(app)
from . import model
@login_manager.user_loader
def load_user(user_id):
    return model.User.query.get(int(user_id))
```

Este fragmento de código inicializa el módulo **`Flask-Login`**, le dice que la vista con el formulario de autenticación es **`auth.login`** y, finalmente, le dice cómo cargar un objeto de usuario de la base de datos dado su identificador.

Para que los objetos de usuario sean compatibles con el módulo **`Flask-Login`**, estos deben heredar también de la clase **`UserMixin`**. Haz este cambio en **`model.py`**:

```
# With the other imports at the beginning
import flask_login
(...)

class User(flask_login.UserMixin, db.Model):
    (...)
```

Para recordar la identidad del usuario, ve a la función de autenticación en **`auth.py`** y haz que comunique la identidad del usuario al módulo **`Flask-Login`**:

```
# With the other imports at the beginning
import flask_login
(...)

if user and bcrypt.check_password_hash(user.password, password):
    # The user exists and the password is correct
    flask_login.login_user(user)
    return redirect(url_for("main.index"))
(...)
```

Puedes obtener la identidad del usuario autenticado en las plantillas a través de la variable **`current_user`**. Esta variable contendrá sus datos si hay un usuario autenticado en ese momento:

```
{% if current_user.is_authenticated: %}
    <div>User: {{current_user.name}}</div>
{% endif %}
```

En controladores, importa **`current_user`** desde **`flask_login`** y la variable **`current_user`** también estará disponible con los datos del usuario autenticado en ese momento.

Modifica **`base.html`** para que todas las páginas muestren el nombre del usuario actual, si lo hay.

Ejercicio 7
Cierre de sesión

Añade un nuevo controlador a **`auth.py`** para cerrar la sesión del usuario actual. Debe indicar al módulo **`Flask-Login`** que cierre la sesión del usuario con la llamada **`flask_login.logout_user()`** y responder con una redirección HTTP a la página de inicio de sesión.

Añade un enlace para cerrar la sesión en **`base.html`**, de forma que los usuarios puedan cerrar la sesión desde cualquier vista. Debe aparecer solo cuando un

usuario esté actualmente autenticado.

Prueba ahora las funciones de inicio y cierre de sesión para comprobar que funcionen correctamente.

Ejercicio 8
Control de acceso

Por último, queremos controlar el acceso a la aplicación para que solo los usuarios autenticados puedan acceder a la vista principal, la vista de perfil de usuario, la vista de mensajes y el controlador de cierre de sesión. Ve a esas funciones de controlador y añade la anotación **`@flask_login.login_required`** inmediatamente antes de su declaración (recuerda importar **`flask_login`** donde sea necesario):

```
@bp.route("/")
@flask_login.login_required
def index():
    (...)
```

Comprueba ahora que, cuando accedes a cualquiera de estos recursos desde tu navegador sin estar autenticado, eres redirigido al formulario de inicio de sesión.

uc3m | Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática