5 Logins de usuário

Resumo: *Nesta quinta parte da série, vamos criar um subsistema de login de usuário.*

Password Hashing

Na sessão anterior, vimos que o modelo de usuário recebeu um campo `password_hash`, que até agora não foi usado. O propósito deste campo é manter um hash da senha do usuário, que será usado para verificar a senha inserida pelo usuário durante o processo de login. O hash de senha é um tópico complicado que deve ser deixado para especialistas em segurança, mas há várias bibliotecas fáceis de usar que implementam toda essa lógica de uma forma que seja simples de ser invocada a partir de um aplicativo.

Um dos pacotes que implementam o hash de senha é o `Werkzeug`, que você pode ter visto referenciado na saída do `pip` quando você instala o `Flask`, já que é uma de suas principais dependências. Como é uma dependência, o `Werkzeug` já está instalado em seu ambiente virtual. A seguinte sessão de *shell* `Python` demonstra como fazer o hash de uma senha com este pacote:

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('bolinhas')
>>> print(hash)
scrypt:32768:8:1$vSaRZKtOdtyo6zjU$ff9155d711a799dc9c35662bc63707aa74f...
```

Neste exemplo, a senha *bolinhas* é transformada em uma longa sequência codificada por meio de uma série de operações criptográficas que não têm operação reversa conhecida, o que significa que uma pessoa que obtém a senha com hash não poderá usá-la para recuperar a senha original. Como medida adicional, se você fizer o hash da mesma senha várias vezes, obterá resultados diferentes, pois todas as senhas com hash recebem um `salt`[1] criptográfico diferente, o que torna impossível identificar se dois usuários têm a mesma senha observando seus `hashes`.

O processo de verificação é feito com uma segunda função do Werkzeug, da seguinte forma:

```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'bolinhas')
```

[1] Um **salt criptográfico** é uma sequência aleatória de caracteres adicionada a uma senha antes de ser criptografada. Em geral, por essa sequência ser produzida por um modelo matemático diferente do hash isso torna o processo de criptografia mais seguro pois aumenta a dificuldade e a identificação do padrão de criptografia.

```
True
>>> check_password_hash(hash, 'bolinha')
False
```

A função de verificação pega um `hash` de senha que foi gerado anteriormente e uma senha inserida pelo usuário no momento do `login`. A função retorna `True` se a senha fornecida pelo usuário corresponde ao `hash`, ou `False` caso contrário.

Toda a lógica de `hash` de senha pode ser implementada como dois novos métodos no modelo de usuário (`app/models.py`):

```
#./app/models.py

from datetime import datetime, timezone
from typing import Optional
import sqlalchemy as sa
import sqlalchemy.orm as so

from flask_login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash

from app import db, login


class User(UserMixin, db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)
    email: so.Mapped[str] = so.mapped_column(sa.String(120), index=True,
                                             unique=True)
    password_hash: so.Mapped[Optional[str]] = so.mapped_column(sa.String(256))

    posts: so.WriteOnlyMapped['Post'] = so.relationship(
        back_populates='author')

    def __repr__(self):
        return '<User {}>'.format(self.username)

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)


@login.user_loader
def load_user(id):
    return db.session.get(User, int(id))
```

```
class Post(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    body: so.Mapped[str] = so.mapped_column(sa.String(140))
    timestamp: so.Mapped[datetime] = so.mapped_column(
        index=True, default=lambda: datetime.now(timezone.utc))
    user_id: so.Mapped[int] = so.mapped_column(sa.ForeignKey(User.id),
                                               index=True)

    author: so.Mapped[User] = so.relationship(back_populates='posts')

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

Com esses dois métodos funcionando, um objeto de usuário agora é capaz de fazer verificação de senha segura, sem a necessidade de armazenar senhas originais. Aqui está um exemplo de uso desses novos métodos:

```
>>> u = User(username='Wanderson', email='wanderson@example.com')
>>> u.set_password('bolinhas')
>>> u.check_password('anotherpassword')
False
>>> u.check_password('bolinhas')
True
```

Introdução ao Flask-Login

Nesta sessão, vamos introduzir a extensão `Flask` chamada `Flask-Login`. Esta extensão gerencia o estado de login do usuário, para que, por exemplo, os usuários possam fazer login no aplicativo e, em seguida, navegar para páginas diferentes enquanto o aplicativo mantém este usuário logado. Ela também fornece a funcionalidade *remember me* que permite que os usuários permaneçam logados mesmo após fechar a janela do navegador. Para estar pronto para esta sessão, precisamos instalar o `Flask-Login` no nosso ambiente virtual:

```
(venv) $ pip install flask-login
```

Assim como tratamos outras extensões, o `Flask-Login` precisa ser criado e inicializado logo após a instância do aplicativo em `app/__init__.py`. É assim que essa extensão é inicializada:

```
#./app/__init__.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

from flask_login import LoginManager

from config import Config
```

```
app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

login = LoginManager(app)
login.login_view = 'login'

from app import routes, models
```

Preparando o modelo de usuário para Flask-Login

A extensão `Flask-Login` funciona com o modelo de usuário do aplicativo e espera que certas propriedades e métodos sejam implementados nele. Essa abordagem é boa, porque, desde que esses itens obrigatórios sejam adicionados ao modelo, o `Flask-Login` não tem nenhum outro requisito, então, por exemplo, ele pode funcionar com modelos de usuário baseados em qualquer sistema de banco de dados.

Os quatro itens obrigatórios são os seguintes:

- `is_authenticated`: uma propriedade que é `True` se o usuário tiver credenciais válidas ou `False` caso contrário.
- `is_active`: uma propriedade que é `True` se a conta do usuário estiver ativa ou `False` caso contrário. `is_anonymous`: uma propriedade que é `False` para usuários regulares e `True` somente para um usuário especial e anônimo.
- `get_id()`: um método que retorna um identificador exclusivo para o usuário como uma *string*.

Podemos implementar esses quatro itens facilmente, mas como as implementações são bastante genéricas, o `Flask-Login` fornece uma classe *mixin* chamada `UserMixin` que inclui implementações seguras que são apropriadas para a maioria das classes de modelo de usuário. Veja como a classe `mixin` é adicionada ao modelo:

```
#./app/models.py

from flask_login import UserMixin

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

Função de carregador de usuário

O `Flask-Login` rastreia o usuário logado armazenando seu identificador exclusivo na sessão de usuário do `Flask`, um espaço de armazenamento atribuído a cada usuário que se conecta ao aplicativo. Cada vez que o usuário logado navega para uma nova página, o `Flask-Login` recupera o `ID` do usuário da sessão e, em seguida, carrega esse usuário na memória.

Como o `Flask-Login` não sabe nada sobre bancos de dados, ele precisa da ajuda do aplicativo para carregar um usuário. Por esse motivo, a extensão espera que o aplicativo configure uma função

de carregador de usuário, que pode ser chamada para carregar um usuário dado o `ID`. Esta função pode ser adicionada no módulo `app/models.py`:

```
#./app/models.py

from app import login
# ...

@login.user_loader
def load_user(id):
    return db.session.get(User, int(id))
```

O carregador de usuário é registrado com `Flask-Login` com o decorador `@login.user_loader`. O `id` que `Flask-Login` passa para a função como um argumento será uma *string*, então bancos de dados que usam `IDs` não-numéricos como expressão de saída precisam converter a *string* para inteiro como você vê acima.

Efetuando login de usuários

Agora, precisamos revisar a função de visualização de `login`, que, como você se lembra, implementou um login falso que apenas emitiu uma mensagem `flash()`. Agora que o aplicativo tem acesso a um banco de dados de usuários e sabe como gerar e verificar `hashes` de senha, essa função de visualização pode ser concluída.

```
#./app/routes.py

from urllib.parse import urlsplit
from flask import render_template, flash, redirect, url_for, request
from flask_login import login_user, logout_user, current_user, login_required
import sqlalchemy as sa
from app import app, db
from app.forms import LoginForm, RegistrationForm
from app.models import User


@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Wanderson'}
    posts = [
        {
            'author': {'username': 'Joao'},
            'body': 'Belo dia em Vila Velha!'
        },
        {
            'author': {'username': 'Maria'},
            'body': 'Bora para o cinema hoje?'
        }
    ]
```

```
    return render_template('index.html', title='Home', user=user, posts=posts)


@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = db.session.scalar(
            sa.select(User).where(User.username == form.username.data))
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        next_page = request.args.get('next')
        if not next_page or urlsplit(next_page).netloc != '':
            next_page = url_for('index')
        return redirect(next_page)
    return render_template('login.html', title='Sign In', form=form)
```

As duas primeiras linhas na função `login()` lidam com uma situação estranha. Imagine que você tem um usuário logado, e o usuário navega para a `URL /login` do seu aplicativo. Claramente isso é um erro, então podemos permitir isso. A variável `current_user` vem do `Flask-Login`, e pode ser usada a qualquer momento durante o tratamento de uma solicitação para obter o objeto de usuário que representa o cliente dessa solicitação. O valor dessa variável pode ser um objeto de usuário do banco de dados (que o `Flask-Login` lê através do retorno de chamada do carregador de usuário que fornecemos acima), ou um objeto de usuário anônimo especial se o usuário ainda não tiver logado. Lembra daquelas propriedades que o `Flask-Login` exigiu no objeto de usuário? Uma delas era `is_authenticated`, que é útil para verificar se o usuário está logado ou não. Quando o usuário já está logado, nós apenas o redirecionamos para a página de índice.

No lugar da chamada `flash()` que usamos antes, agora podemos logar o usuário de verdade. O primeiro passo é carregar o usuário do banco de dados. O nome de usuário veio com o envio do formulário, então podemos consultar o banco de dados com ele para encontrar o usuário. Para esse propósito, estamos usando a cláusula `where()`, para encontrar usuários com o nome de usuário fornecido. Como sabemos que haverá apenas um ou zero resultados, executamos a consulta chamando `db.session.scalar()`, que retornará o objeto do usuário se ele existir, ou `None` se não existir. No sessão anterior, vimos que quando chamamos o método `all()` a consulta é executada e obtemos uma lista de todos os resultados que correspondem a essa consulta. O método `first()` é outra maneira comumente usada para executar uma consulta, quando só precisamos ter um resultado.

Se obtivermos uma correspondência para o nome de usuário fornecido, podemos verificar se a senha que também veio com o formulário é válida. Isso é feito invocando o método `check_password()` que definimos acima. Isso pegará o `hash` da senha armazenada com o usuário e determinará se a senha inserida no formulário corresponde ao `hash` ou não. Então agora temos duas possíveis condições de erro: o nome de usuário pode ser inválido, ou a senha pode estar incorreta para o usuário. Em qualquer um desses casos, mostramos uma mensagem e o redirecionamos de volta para o `prompt` de `login` para que o usuário possa tentar novamente.

Se o nome de usuário e a senha estiverem corretos, então chamamos a função `login_user()`,

que vem do `Flask-Login`. Esta função registrará o usuário como logado, o que significa que qualquer página futura para a qual o usuário navegue terá a variável `current_user` definida para esse usuário.

Para concluir o processo de login, apenas redirecionamos o usuário recém-logado para a página de índice.

Desconectando usuários

Também precisamos oferecer aos usuários a opção de fazer `logout` do aplicativo. Isso pode ser feito com a função `logout_user()` do `Flask-Login`. Aqui está a função de visualização de `logout`:

```
#./app/routes.py

# ...
from flask_login import logout_user

# ...

@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))
```

Para expor esse *link* aos usuários, podemos fazer com que o *link* `Login` na barra de navegação mude automaticamente para um *link* `Logout` depois que o usuário fizer `login`. Isso pode ser feito com uma condicional no modelo `base.html`:

```
<!-- app/templates/base.html -->

<!doctype html>
<html>
    <head>
        {% if title %}
        <title>{{ title }} - UVV</title>
        {% else %}
        <title>Bem vindo ao Blog UVV</title>
        {% endif %}
    </head>
    <body>
        <div>
            Microblog:
            <a href="{{ url_for('index') }}">Home</a>
            {% if current_user.is_anonymous %}
            <a href="{{ url_for('login') }}">Login</a>
            {% else %}
            <a href="{{ url_for('logout') }}">Logout</a>
            {% endif %}
        </div>
        <hr>
        {% with messages = get_flashed_messages() %}
```

```
        {% if messages %}
        <ul>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
        </ul>
        {% endif %}
        {% endwith %}
        {% block content %}{% endblock %}
    </body>
</html>
```

A propriedade `is_anonymous` é um dos atributos que o `Flask-Login` adiciona aos objetos de usuário por meio da classe `UserMixin`. A expressão `current_user.is_anonymous` será `True` somente quando o usuário não estiver logado.

Exigindo que os usuários façam login

O `Flask-Login` fornece um recurso muito útil que força os usuários a efetuar `login` antes de poderem visualizar certas páginas do aplicativo. Se um usuário que não esteja logado tentar visualizar uma página protegida, o `Flask-Login` redirecionará automaticamente o usuário para o formulário de `login` e somente redirecionará de volta para a página que o usuário queria visualizar após o processo de login ser concluído.

Para que esse recurso seja implementado, o `Flask-Login` precisa saber qual é a função de visualização que manipula `logins`. Isso pode ser adicionado em `app/__init__.py`:

```
#./app/__init__.py

# ...
login = LoginManager(app)
login.login_view = 'login'
```

O valor **'login'** acima é o nome da função (ou *endpoint*) para a visualização de `login`. Em outras palavras, o nome que você usaria em uma chamada `url_for()` para obter a `URL`.

A maneira como o `Flask-Login` protege uma função de visualização contra usuários anônimos é com um decorador chamado `@login_required`. Quando você adiciona esse decorador a uma função de visualização abaixo do decorador `@app.route` do `Flask`, a função se torna protegida e não permitirá acesso a usuários que não sejam autenticados. Aqui está como o decorador pode ser aplicado à função de visualização de índice do aplicativo:

```
#./app/routes.py

from flask_login import login_required

@app.route('/')
@app.route('/index')
@login_required
def index():
```

```
    # ...
```

O que resta é implementar o redirecionamento de volta do `login` bem-sucedido para a página que o usuário queria acessar. Quando um usuário que não está logado acessa uma função de visualização protegida com o decorador `@login_required`, o decorador vai redirecionar para a página de login, mas vai incluir algumas informações extras neste redirecionamento para que o aplicativo possa então retornar à página original. Se o usuário navegar para `/index`, por exemplo, o decorador `@login_required` interceptará a solicitação e responderá com um redirecionamento para `/login`, mas adicionará um argumento de string de consulta a esta `URL`, tornando a `URL` de redirecionamento completa `/login?next=/index`. O próximo argumento de *string* de consulta é definido para a `URL` original, para que o aplicativo possa usá-lo para redirecionar de volta após o login.

Aqui está um trecho de código que mostra como ler e processar o próximo argumento de *string* de consulta. As alterações estão nas quatro linhas abaixo da chamada `login_user()`.

```
#./app/routes.py --> Redirect to \"next\" page

from flask import request
from urllib.parse import urlsplit

@app.route('/login', methods=['GET', 'POST'])
def login():
    # ...
    if form.validate_on_submit():
        user = db.session.scalar(
            sa.select(User).where(User.username == form.username.data))
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        next_page = request.args.get('next')
        if not next_page or urlsplit(next_page).netloc != '':
            next_page = url_for('index')
        return redirect(next_page)
    # ...
```

Logo após o usuário efetuar `login` chamando a função `login_user()` do `Flask-Login`, o valor do próximo argumento da *string* de consulta é obtido. O `Flask` fornece uma variável de solicitação que contém todas as informações que o cliente enviou com a solicitação. Em particular, o atributo `request.args` expõe o conteúdo da *string* de consulta em um formato de dicionário amigável. Na verdade, há três casos possíveis que precisam ser considerados para determinar para onde redirecionar após um `login` bem-sucedido:

- Se a `URL` de `login` não tiver um próximo argumento, o usuário será redirecionado para a página de índice.
- Se a `URL` de `login` incluir um próximo argumento definido como um caminho relativo (ou em outras palavras, uma `URL` sem a parte do domínio), o usuário será redirecionado para essa `URL`.
- Se a `URL` de `login` incluir um próximo argumento definido como uma `URL` completa que inclua

um nome de domínio, essa `URL` será ignorada e o usuário será redirecionado para a página de índice.

O primeiro e o segundo casos são autoexplicativos. O terceiro caso está em vigor para tornar o aplicativo mais seguro. Um invasor pode inserir uma `URL` para um site malicioso no próximo argumento, então o aplicativo redireciona somente quando a `URL` é relativa, o que garante que o redirecionamento permaneça dentro do mesmo site que o aplicativo. Para determinar se a `URL` é absoluta ou relativa, eu a analiso com a função `urlsplit()` do `Python` e então verificamos se o componente `netloc`‘ está definido ou não.