6 Página de perfil e avatares

Resumo: *Esta é a sexta parte da sequência do nosso desenvolvimento web, na qual vamos criar a página de perfil do usuário.*

Página de perfil do usuário

Para criar uma página de perfil de usuário, vamos adicionar uma rota `/user/<username>` ao aplicativo.

```
@app.route('/user/<username>')
@login_required
def user(username):
    user = db.first_or_404(sa.select(User).where(User.username == username))
    posts = [
        {'author': user, 'body': 'Test post #1'},
        {'author': user, 'body': 'Test post #2'}
    ]
    return render_template('user.html', user=user, posts=posts)
```

O decorador `@app.route` que usamos para declarar esta função de visualização parece um pouco diferente dos anteriores. Neste caso, temos um componente dinâmico nele, que é indicado como o componente de URL `<username>` que é cercado por `<` e `>`. Quando uma rota tem um componente dinâmico, o `Flask` aceitará qualquer texto nessa parte da URL e invocará a função de visualização com o texto real como argumento. Por exemplo, se o navegador do cliente solicitar a URL `/user/wanderson`, a função de visualização será chamada com o argumento username definido como '`wanderson`'. Esta função de visualização só será acessível a usuários logados, então adicionamos o decorador `@login_required` do `Flask-Login`.

A implementação desta função de visualização é bastante simples. Primeiro, tento carregar o usuário do banco de dados usando uma consulta pelo username. Vimos antes que uma consulta de banco de dados pode ser executada com `db.session.scalars()` se desejamos obter todos os resultados, ou `db.session.scalar()` se você quiser obter apenas o primeiro resultado ou `None` se não houver resultados. Nesta função de exibição, estamos usando uma variante de `scalar()` que é fornecida pelo `Flask-SQLAlchemy` chamada `db.first_or_404()`, que funciona como `scalar()` quando há resultados, mas no caso de não haver resultados, ele envia automaticamente um erro

404 de volta para o cliente. Ao executar a consulta dessa forma, poupamos verificar se a consulta retornou um usuário, porque quando o nome de usuário não existe no banco de dados, a função não retornará e, em vez disso, uma exceção 404 será gerada.

Se a consulta de banco de dados não disparar um erro 404, isso significa que um usuário com o nome de usuário fornecido foi encontrado. Em seguida, inicializamos uma lista falsa de posts para esse usuário e renderizamos um novo template `user.html` para o qual passamos o objeto `user` e a lista de `posts`.

O template `user.html` é mostrado abaixo:

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

{% extends "base.html" %}

{% block content %}
    <h1>User: {{ user.username }}</h1>
    <hr>
    {% for post in posts %}
    <p>
    {{ post.author.username }} says: <b>{{ post.body }}</b>
    </p>
    {% endfor %}
{% endblock %}
```

A página de perfil agora está completa, mas um *link* para ela não existe em nenhum lugar do site. Para tornar um pouco mais fácil para os usuários verificarem seus próprios perfis, vou adicionar um *link* para ele na barra de navegação no topo:

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

        <div>
            UVV:
            <a href="{{ url_for('index') }}">Home</a>
            {% if current_user.is_anonymous %}
            <a href="{{ url_for('login') }}">Login</a>
            {% else %}
            <a href="{{ url_for('user', username=current_user.username) }}">↩
                Profile</a>
            <a href="{{ url_for('logout') }}">Logout</a>
            {% endif %}
        </div>
```

A única mudança interessante aqui é a chamada `url_for()` que é usada para gerar o *link* para a página de perfil. Como a função de visualização do perfil do usuário recebe um argumento dinâmico, a função `url_for()` recebe um valor para esta parte da URL como um argumento de palavra-chave. Como este é um link que aponta para o perfil do usuário logado, posso usar o `current_user` do `Flask-Login` para gerar a URL correta.

Experimente o aplicativo agora. Clicar no *link* `Profile` no topo deve levá-lo para sua própria página de usuário. Neste ponto, não há *links* que o levarão para a página de perfil de outros usuários,

mas se você quiser acessar essas páginas, você pode digitar a URL manualmente na barra de endereços do navegador. Por exemplo, se você tiver um usuário chamado "joao"registrado em seu aplicativo, você pode visualizar o perfil de usuário correspondente digitando `http://localhost:5000/user/joao` na barra de endereços.

Avatars

Tenho certeza de que você concorda que as páginas de perfil que acabamos de criar são bem chatas. Para torná-las um pouco mais interessantes, vamos adicionar avatares de usuários, mas em vez de ter que lidar com uma coleção possivelmente grande de imagens carregadas no servidor, vamos usar o serviço **Gravatar** para fornecer imagens para todos os usuários.

O serviço Gravatar é muito simples de usar. Para solicitar uma imagem para um determinado usuário, uma URL com o formato `https://www.gravatar.com/avatar/<hash>` deve ser usada, onde `<hash>` é o **hash MD5** do endereço de e-mail do usuário. Abaixo, você pode ver como obter a URL do Gravatar para um usuário com e-mail joao@example.com:

```
>>> from hashlib import md5
>>> 'https://www.gravatar.com/avatar/' + md5(b'joao@example.com').hexdigest()
'https://www.gravatar.com/avatar/dad69301665622b3fd122123053a2eb7'
```

Outro argumento interessante que pode ser passado para o Gravatar como um argumento de *string* de consulta é `d`, que determina qual imagem o Gravatar fornece para usuários que não têm um avatar registrado no serviço. Meu favorito é chamado "`identicon`", que retorna um belo design geométrico que é diferente para cada e-mail. Por exemplo:

Observe que algumas extensões de privacidade do navegador da web, como o Ghostery, bloqueiam imagens do Gravatar, pois consideram que a Automattic (os proprietários do serviço Gravatar) podem determinar quais sites você visita com base nas solicitações que recebem para seu avatar. Se você não vir avatares no seu navegador, considere que o problema pode ser devido a uma extensão que você instalou no seu navegador.

Como os avatares são associados aos usuários, faz sentido adicionar a lógica que gera as URLs do avatar ao modelo do usuário.

```
from hashlib import md5
# ...

class User(UserMixin, db.Model):
    # ...
    def avatar(self, size):
        digest = md5(self.email.lower().encode('utf-8')).hexdigest()
        return f'https://www.gravatar.com/avatar/{digest}?d=identicon&s={size}'
```

O novo método `avatar()` da classe `User` retorna a URL da imagem do avatar do usuário, dimensionada para o tamanho solicitado em *pixels*. Para usuários que não têm um avatar registrado, uma imagem "identicon" será gerada. Para gerar o `hash` `MD5`, primeiro convertemos o e-mail para letras minúsculas, pois isso é exigido pelo serviço Gravatar. Então, como o suporte MD5 em `Python` funciona em *bytes* e não em *strings*, codifico a *string* como *bytes* antes de passá-la para a função `hash`.

Se você estiver interessado em aprender sobre outras opções oferecidas pelo serviço Gravatar, visite o site de documentação.

O próximo passo é inserir as imagens de avatar no modelo de perfil do usuário:

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

{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
    <p>
    {{ post.author.username }} says: <b>{{ post.body }}</b>
    </p>
    {% endfor %}
{% endblock %}
```

O legal de tornar a classe `User` responsável por retornar URLs de avatar é que se algum dia decidirmos que avatares do Gravatar não são o que desejamos, podemos simplesmente reescrever o método `avatar()` para retornar URLs diferentes, e todos os modelos começarão a mostrar os novos avatares automaticamente.

Agora temos um avatar grande e bonito no topo da página de perfil do usuário, mas realmente não há razão para parar por aí. Temos algumas postagens do usuário na parte inferior que poderiam ter um pequeno avatar também. Para a página de perfil do usuário, é claro, todas as postagens terão o mesmo avatar, mas então podemos implementar a mesma funcionalidade na página principal, e então cada postagem será decorada com o avatar do autor, e isso ficará muito legal.

Para mostrar avatares para as postagens individuais, só precisamos fazer mais uma pequena alteração no modelo:

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

{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
```

```
        </tr>
    </table>
    <hr>
    {% for post in posts %}
    <table>
        <tr valign="top">
            <td><img src="{{ post.author.avatar(36) }}"></td>
            <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
        </tr>
    </table>
    {% endfor %}
{% endblock %}
```

Usando sub-modelos Jinja

Projetamos a página de perfil do usuário para que ela exiba as postagens escritas pelo usuário, junto com seus avatares. Agora queremos que a página de índice também exiba postagens com um layout similar. Poderíamos simplesmente copiar/colar a parte do modelo que lida com a renderização de uma postagem, mas isso não é realmente o ideal porque mais tarde, se decidirmos fazer alterações neste *layout*, teremos que lembrar de atualizar ambos os modelos.

Em vez disso, vamos criar um submodelo que renderiza apenas uma postagem e, em seguida, vamos referenciá-lo a partir dos modelos `user.html` e `index.html`. Para começar, posso criar o submodelo, apenas com a marcação HTML para uma única postagem. Vamos nomear este modelo `app/templates/_post.html`. O prefixo `_` é apenas uma convenção de nomenclatura para me ajudar a reconhecer quais arquivos de modelo são submodelos.

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

    <table>
        <tr valign="top">
            <td><img src="{{ post.author.avatar(36) }}"></td>
            <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
        </tr>
    </table>
```

Para invocar este submodelo do modelo `user.html`, uso a instrução include do `Jinja`:

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

{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
```

```
    <hr>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
{% endblock %}
```

A página de índice do aplicativo ainda não está totalmente definida, então não vou adicionar essa funcionalidade lá ainda.

Perfis mais interessantes

Um problema que as novas páginas de perfil de usuário têm é que elas não mostram muita coisa. Os usuários gostam de contar um pouco sobre eles nessas páginas, então vou deixá-los escrever algo sobre si mesmos para mostrar aqui. Também vou manter o controle da última vez que cada usuário acessou o site e também mostrar isso na página de perfil deles.

A primeira coisa que preciso fazer para dar suporte a todas essas informações extras é estender a tabela de usuários no banco de dados com dois novos campos:

```
#   app/models.py

class User(UserMixin, db.Model):
    # ...
    about_me: so.Mapped[Optional[str]] = so.mapped_column(sa.String(140))
    last_seen: so.Mapped[Optional[datetime]] = so.mapped_column(
        default=lambda: datetime.now(timezone.utc))
```

Toda vez que o banco de dados é modificado é necessário gerar uma migração de banco de dados. Já mostramos como configurar o aplicativo para rastrear alterações no banco de dados por meio de *scripts* de migração. Agora temos dois novos campos que queremos adicionar ao banco de dados, então o primeiro passo é gerar o *script* de migração:

```
(venv) $ flask db migrate -m "new fields in user model"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added column 'user.about_me'
INFO  [alembic.autogenerate.compare] Detected added column 'user.last_seen'
  Generating migrations/versions/37f06a334dbf_new_fields_in_user_model.py ... ↩
     done
```

A saída do comando migrate parece boa, pois mostra que os dois novos campos na classe User foram detectados. Agora posso aplicar essa alteração ao banco de dados:

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 780739b227a7 -> 37f06a334dbf,↩
     new fields in user model
```

Perceba o quão útil é trabalhar com uma estrutura de migração. Todos os usuários que estavam no banco de dados ainda estão lá, a estrutura de migração aplica cirurgicamente as alterações no *script* de migração sem destruir nenhum dado.

Para a próxima etapa, adicionaremos esses dois novos campos ao modelo de perfil do usuário:

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

{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td>
                <h1>User: {{ user.username }}</h1>
                {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
                {% if user.last_seen %}<p>Last seen on: {{ user.last_seen }}</p↩
                    >{% endif %}
            </td>
        </tr>
    </table>
    ...
{% endblock %}
```

Note que estou envolvendo esses dois campos nas condicionais do `Jinja`, porque eu só quero que eles sejam visíveis se estiverem definidos. Neste ponto, esses dois novos campos estão vazios para todos os usuários, então você não verá esses campos ainda.

Registrando o horário da última visita de um usuário

Vamos começar com o campo `last_seen`, que é o mais fácil dos dois. O que eu quero fazer é escrever o horário atual neste campo para um determinado usuário sempre que esse usuário enviar uma solicitação ao servidor.

Adicionar o `login` para definir este campo em cada função de visualização possível que pode ser solicitada do navegador é obviamente impraticável, mas executar um pouco de lógica genérica antes de uma solicitação ser despachada para uma função de visualização é uma tarefa tão comum em aplicativos da web que o `Flask` a oferece como um recurso nativo. Dê uma olhada na solução:

```
#   app/routes.py

from datetime import datetime, timezone

@app.before_request
def before_request():
    if current_user.is_authenticated:
        current_user.last_seen = datetime.now(timezone.utc)
        db.session.commit()
```

O decorador `@before_request` do `Flask` registra a função decorada para ser executada logo antes da função de visualização. Isso é extremamente útil porque agora podemos inserir o código que desejo executar antes de qualquer função de visualização no aplicativo e podemos também tê-lo em um único lugar. A implementação simplesmente verifica se o `current_user` está conectado e, nesse caso, define o campo `last_seen` para o horário atual. Como mencionado antes, um aplicativo de servidor precisa trabalhar em unidades de tempo consistentes e a prática padrão é usar o fuso horário UTC. Usar o horário local do sistema não é uma boa ideia, porque o que entra no banco de dados depende da sua localização.

A última etapa é confirmar a sessão do banco de dados, para que a alteração feita acima seja gravada no banco de dados. Se você está se perguntando por que não há `db.session.add()` antes do `commit`, considere que quando você referencia `current_user`, o `Flask-Login` invocará a função de retorno de chamada do carregador de usuário, que executará uma consulta ao banco de dados que colocará o usuário de destino na sessão do banco de dados. Então você pode adicionar o usuário novamente nesta função, mas não é necessário porque ele já está lá.

Se você visualizar sua página de perfil depois de fazer essa alteração, verá a linha "Last seen on" com um horário muito próximo do horário atual. E se você navegar para fora da página de perfil e retornar, verá que o horário é constantemente atualizado.

O fato de estarmos armazenando esses carimbos de data/hora no fuso horário UTC faz com que o horário exibido na página de perfil também esteja em UTC. Além disso, o formato do horário não é o que você esperaria, pois é, na verdade, a representação interna do objeto `datetime` do `Python`. Por enquanto, não vamos nos preocupar com esses dois problemas, pois abordaremos o tópico de manipulação de datas e horários em um aplicativo da web posteriormente.

Editor de perfil

Também precisamos dar aos usuários um formulário no qual eles possam inserir algumas informações sobre si mesmos. O formulário permitirá que os usuários alterem seus nomes de usuário e também escrevam algo sobre si mesmos, para ser armazenado no novo campo `about_me`. Vamos começar a escrever uma classe de formulário para ele:

```
#   app/forms.py

from wtforms import TextAreaField
from wtforms.validators import Length

# ...

class EditProfileForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
    submit = SubmitField('Submit')
```

Estamos usando um novo tipo de campo e um novo validador neste formulário. Para o campo "About me", estamos usando um `TextAreaField`, que é uma caixa multilinha na qual o usuário pode inserir texto. Para validar este campo, estou usando *Length*, que garantirá que o texto inserido esteja entre 0 e 140 caracteres, que é o espaço alocado para o campo correspondente no banco de dados.

O modelo que renderiza este formulário é mostrado abaixo:

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

{% extends "base.html" %}

{% block content %}
    <h1>Edit Profile</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.about_me.label }}<br>
            {{ form.about_me(cols=50, rows=4) }}<br>
            {% for error in form.about_me.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

E, finalmente, aqui está a função de visualização que une tudo:

```
#   app/routes.py

from app.forms import EditProfileForm

@app.route('/edit_profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.username = form.username.data
        current_user.about_me = form.about_me.data
        db.session.commit()
        flash('Your changes have been saved.')
        return redirect(url_for('edit_profile'))
    elif request.method == 'GET':
        form.username.data = current_user.username
        form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', title='Edit Profile',
                           form=form)
```

Esta função de visualização processa o formulário de uma forma ligeiramente diferente. Se `validate_on_submit()` retornar `True`, copiamos os dados do formulário para o objeto do usuário e, em seguida, escrevemos o objeto no banco de dados. Mas quando `validate_on_submit()` retornar `False`, pode ser devido a dois motivos diferentes. Primeiro, pode ser porque o navegador acabou de enviar uma solicitação `GET`, que preciso responder fornecendo uma versão inicial do modelo de formulário. Também pode ser quando o navegador envia uma solicitação `POST` com dados do formulário, mas algo nesses dados é inválido. Para este formulário, preciso tratar esses dois casos separadamente. Quando o formulário está sendo solicitado pela primeira vez com uma solicitação `GET`, quero preencher previamente os campos com os dados armazenados no banco de dados, então precisamos fazer o inverso do que fiz no caso de envio e mover os dados armazenados nos campos do usuário para o formulário, pois isso garantirá que esses campos do formulário tenham os dados atuais armazenados para o usuário. Mas no caso de um erro de validação, não queremos escrever nada nos campos do formulário, porque eles já foram preenchidos pelo `WTForms`. Para distinguir entre esses dois casos, verificamos `request.method`, que será `GET` para a solicitação inicial e `POST` para um envio que falhou na validação.

Para facilitar o acesso dos usuários à página do editor de perfil, posso adicionar um link na página de perfil deles:

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

                {% if user == current_user %}
                <p><a href="{{ url_for('edit_profile') }}">Edit your profile</a↩
                    ></p>
                {% endif %}
```

Preste atenção à condicional inteligente que estamos usando para garantir que o *link* Editar apareça quando você estiver visualizando seu próprio perfil, mas não quando estiver visualizando o perfil de outra pessoa.