The Flask Mega-Tutorial, Part VI: Profile Page and Avatars

Posted by Miguel Grinberg *on* December 3, 2023 *under* Python Flask Programming

This is the sixth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to create the user profile page.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from Amazon. Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it here.

For your reference, here is the complete list of articles in this series:

- Chapter 1: Hello, World!
- Chapter 2: Templates
- Chapter 3: Web Forms
- Chapter 4: Database
- Chapter 5: User Logins
- Chapter 6: Profile Page and Avatars (this article)
- Chapter 7: Error Handling
- Chapter 8: Followers
- Chapter 9: Pagination
- Chapter 10: Email Support
- Chapter 11: Facelift
- Chapter 12: Dates and Times
- Chapter 13: I18n and L10n
- Chapter 14: Ajax
- Chapter 15: A Better Application Structure
- Chapter 16: Full-Text Search
- Chapter 17: Deployment on Linux
- Chapter 18: Deployment on Heroku
- Chapter 19: Deployment on Docker Containers
- Chapter 20: Some JavaScript Magic
- Chapter 21: User Notifications

This chapter is going to be dedicated to adding user profile pages to the application. A user profile page is a page in which information about a user is presented, often with information entered by the users themselves. I will show you how to generate profile pages for all users dynamically, and then I'll add a small profile editor that users can use to enter their information.

The GitHub links for this chapter are: [Browse](#), [Zip](#), [Diff](#).

User Profile Page

To create a user profile page, let's add a */user/<username>* route to the application.

---

app/routes.py: User profile view function

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

The `@app.route` decorator that I used to declare this view function looks a little bit different than the previous ones. In this case I have a dynamic component in it, which is indicated as the `<username>` URL component that is surrounded by `<` and `>`. When a route has a dynamic component, Flask will accept any text in that portion of the URL, and will invoke the view function with the actual text as an argument. For example, if the client browser requests URL `/user/susan`, the view function is going to be called with the argument `username` set to `'susan'`. This view function is only going to be accessible to logged-in users, so I have added the `@login_required` decorator from Flask-Login.

The implementation of this view function is fairly simple. I first try to load the user from the database using a query by the username. You have seen before that a database query can be executed with

`db.session.scalars()` if you want to get all results, or `db.session.scalar()` if you want to get just the first result or `None` if there are zero results. In this view function I'm using a variant of `scalar()` that is provided by Flask-SQLAlchemy called `db.first_or_404()`, which works like `scalar()` when there are results, but in the case that there are no results it automatically sends a [404 error](#) back to the client. By executing the query in this way I save myself from checking if the query returned a user, because when the username does not exist in the database the function will not return and instead a 404 exception will be raised.

If the database query does not trigger a 404 error, then that means that a user with the given username was found. Next I initialize a fake list of posts for this user, and render a new *user.html* template to which I pass the user object and the list of posts.

The *user.html* template is shown below:

app/templates/user.html: User profile template

```
{% 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 %}
```

The profile page is now complete, but a link to it does not exist anywhere in the web site. To make it a bit more easy for users to check their own profile, I'm going to add a link to it in the navigation bar at the top:

app/templates/base.html: User profile template

```
    <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('user', username=current_user.username
        <a href="{{ url_for('logout') }}">Logout</a>
```

```
        {% endif %}
    </div>
```

The only interesting change here is the `url_for()` call that is used to generate the link to the profile page. Since the user profile view function takes a dynamic argument, the `url_for()` function receives a value for this part of the URL as a keyword argument. Because this is a link that points to the logged-in user's profile, I can use Flask-Login's `current_user` to generate the correct URL.
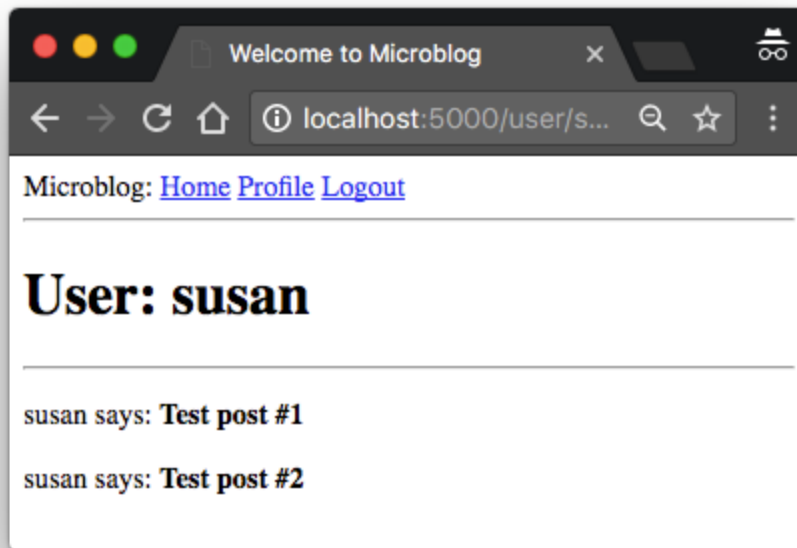
Give the application a try now. Clicking on the `Profile` link at the top should take you to your own user page. At this point there are no links that will take you to the profile page of other users, but if you want to access those pages you can type the URL by hand in the browser's address bar. For example, if you have a user named "john" registered on your application, you can view the corresponding user profile by typing *http://localhost:5000/user/john* in the address bar.

Avatars

I'm sure you agree that the profile pages that I just built are pretty boring. To make them a bit more interesting, I'm going to add user avatars, but instead of having to deal with a possibly large collection of uploaded

images in the server, I'm going to use the [Gravatar](#) service to provide images for all users.

The Gravatar service is very simple to use. To request an image for a given user, a URL with the format *https://www.gravatar.com/avatar/<hash>* must be used, where `<hash>` is the MD5 hash of the user's email address. Below you can see how to obtain the Gravatar URL for a user with email `john@example.com`:

```
>>> from hashlib import md5
>>> 'https://www.gravatar.com/avatar/' + md5(b'john@example.com').hexd
'https://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6'
```

If you want to see an actual example, my own Gravatar URL is:

```
https://www.gravatar.com/avatar/729e26a2a2c7ff24a71958d4aa4e5f35
```

Here is what Gravatar returns for this URL if you type it in a browser's address bar:
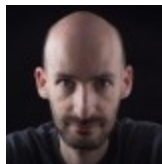
By default, the image size returned is 80x80 pixels, but a different size can be requested by adding a `s` argument to the URL's query string. For example, to obtain my own avatar as a 128x128 pixel image, the URL is \linebreak *https://www.gravatar.com/avatar/729e26a2a2c7ff24a71958d4aa4e5f35?s=128*.

Another interesting argument that can be passed to Gravatar as a query string argument is `d`, which determines what image Gravatar provides for users that do not have an avatar registered with the service. My favorite is called "identicon", which returns a nice geometric design that is different for every email. For example:

Note that some privacy web browser extensions such as Ghostery block Gravatar images, as they consider that Automattic (the owners of the Gravatar service) can determine what sites you visit based on the requests they get for your avatar. If you don't see avatars in your browser, consider that the problem may be due to an extension that you have installed in your browser.

Since avatars are associated with users, it makes sense to add the logic that generates the avatar URLs to the user model.

app/models.py: User avatar URLs

```python
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&
```

The new `avatar()` method of the `User` class returns the URL of the user's avatar image, scaled to the requested size in pixels. For users that don't have an avatar registered, an "identicon" image will be generated. To generate the MD5 hash, I first convert the email to lower case, as this is required by the Gravatar service. Then, because the MD5 support in Python works on bytes and not on strings, I encode the string as bytes before passing it on to the hash function.

If you are interested in learning about other options offered by the Gravatar service, visit their [documentation website](#).

The next step is to insert the avatar images in the user profile template:

app/templates/user.html: User avatar in template

```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 %}
```

The nice thing about making the `User` class responsible for returning avatar URLs is that if some day I decide Gravatar avatars are not what I want, I can just rewrite the `avatar()` method to return different URLs, and all the templates will start showing the new avatars automatically.

I now have a nice big avatar at the top of the user profile page, but really there is no reason to stop there. I have some posts from the user at the bottom that could each have a little avatar as well. For the user profile page of course all posts will have the same avatar, but then I can implement the same functionality on the main page, and then each post will be decorated with the author's avatar, and that will look really nice.

To show avatars for the individual posts I just need to make one more small change in the template:

app/templates/user.html: User avatars in posts
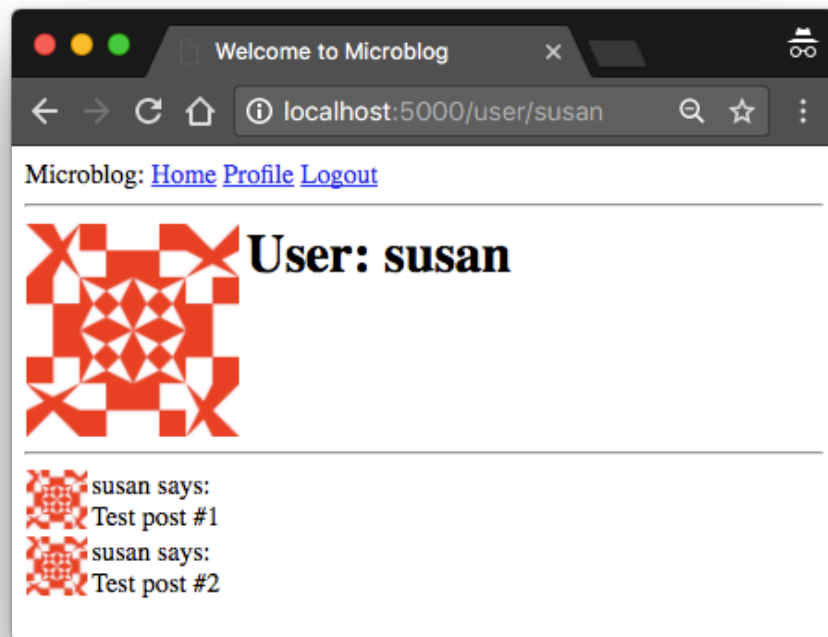
```
{% extends "base.html" %}

{% 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 }}</t
        </tr>
    </table>
    {% endfor %}
{% endblock %}
```

Using Jinja Sub-Templates

I designed the user profile page so that it displays the posts written by the user, along with their avatars. Now I want the index page to also display posts with a similar layout. I could just copy/paste the portion of the template that deals with the rendering of a post, but that is really not ideal because later if I decide to make changes to this layout I'm going to have to remember to update both templates.

Instead, I'm going to make a sub-template that just renders one post, and then I'm going to reference it from both the *user.html* and *index.html* templates. To begin, I can create the sub-template, with just the HTML markup for a single post. I'm going to name this template *app/templates/_post.html*. The _ prefix is just a naming convention to help me recognize which template files are sub-templates.

app/templates/_post.html: Post sub-template

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

```
        </tr>
    </table>
```

To invoke this sub-template from the *user.html* template I use Jinja's `include` statement:

---

app/templates/user.html: User avatars in posts

```
{% 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 %}
```

The index page of the application isn't really fleshed out yet, so I'm not going to add this functionality there yet.

More Interesting Profiles

One problem the new user profile pages have is that they don't really show much on them. Users like to tell a bit about them on these pages, so I'm going to let them write something about themselves to show here. I'm also going to keep track of what was the last time each user accessed the site and also show display it on their profile page.

The first thing I need to do to support all this extra information is to extend the `users` table in the database with two new fields:

---

app/models.py: New fields in user model

```
class User(UserMixin, db.Model):
    # ...
    about_me: so.Mapped[Optional[str]] = so.mapped_column(sa.String(14
    last_seen: so.Mapped[Optional[datetime]] = so.mapped_column(
        default=lambda: datetime.now(timezone.utc))
```

Every time the database is modified it is necessary to generate a database migration. In [Chapter 4](#) I showed you how to set up the application to track database changes through migration scripts. Now I have two new fields that I want to add to the database, so the first step is to generate the migration script:

```
(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
INFO  [alembic.autogenerate.compare] Detected added column 'user.last_
  Generating migrations/versions/37f06a334dbf_new_fields_in_user_model
```

The output of the `migrate` command looks good, as it shows that the two new fields in the `User` class were detected. Now I can apply this change to the database:

```
(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 -> 37f0
```

I hope you realize how useful it is to work with a migration framework. Any users that were in the database are still there, the migration framework surgically applies the changes in the migration script without destroying any data.

For the next step, I'm going to add these two new fields to the user profile template:

app/templates/user.html: Show user information in user profile template

```
{% 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>{% end
                {% if user.last_seen %}<p>Last seen on: {{ user.last_s
            </td>
        </tr>
```

```
        </table>
        ...
{% endblock %}
```

Note that I'm wrapping these two fields in Jinja's conditionals, because I only want them to be visible if they are set. At this point these two new fields are empty for all users, so you are not going to see these fields yet.

Recording The Last Visit Time For a User

Let's start with the `last_seen` field, which is the easier of the two. What I want to do is write the current time on this field for a given user whenever that user sends a request to the server.

Adding the login to set this field on every possible view function that can be requested from the browser is obviously impractical, but executing a bit of generic logic ahead of a request being dispatched to a view function is such a common task in web applications that Flask offers it as a native feature. Take a look at the solution:

app/routes.py: Record time of last visit

```python
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()
```

The `@before_request` decorator from Flask register the decorated function to be executed right before the view function. This is extremely useful because now I can insert code that I want to execute before any view function in the application, and I can have it in a single place. The implementation simply checks if the `current_user` is logged in, and in that case sets the `last_seen` field to the current time. I mentioned this before, a server application needs to work in consistent time units, and the standard practice is to use the UTC time zone. Using the local time of the system is not a good idea, because then what goes in the database is dependent on your location.

The last step is to commit the database session, so that the change made above is written to the database. If you are wondering why there is

no `db.session.add()` before the commit, consider that when you reference `current_user`, Flask-Login will invoke the user loader callback function, which will run a database query that will put the target user in the database session. So you can add the user again in this function, but it is not necessary because it is already there.

If you view your profile page after you make this change, you will see the "Last seen on" line with a time that is very close to the current time. And if you navigate away from the profile page and then return, you will see that the time is constantly updated.

The fact that I'm storing these timestamps in the UTC timezone makes the time displayed on the profile page also be in UTC. In addition to that, the format of the time is not what you would expect, since it is actually the internal representation of the Python `datetime` object. For now, I'm not going to worry about these two issues, since I'm going to address the topic of handling dates and times in a web application in a later chapter.
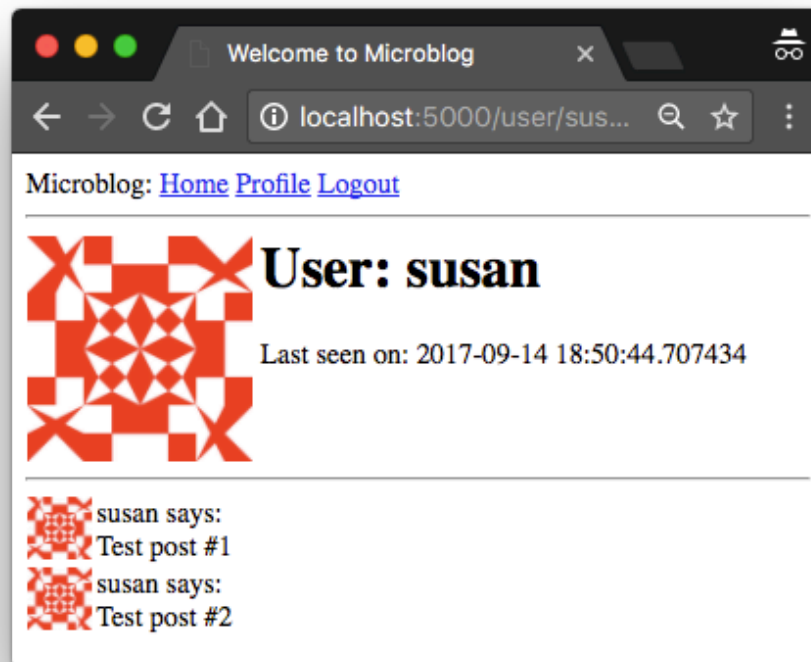
Profile Editor

I also need to give users a form in which they can enter some information about themselves. The form is going to let users change their username, and also write something about themselves, to be stored in the new `about_me` field. Let's start writing a form class for it:

---

app/forms.py: Profile editor form

```python
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
    submit = SubmitField('Submit')
```

I'm using a new field type and a new validator in this form. For the "About" field I'm using a `TextAreaField`, which is a multi-line box in which the user can enter text. To validate this field I'm using `Length`, which will make sure that the text entered is between 0 and 140 characters, which is the space I have allocated for the corresponding field in the database.

The template that renders this form is shown below:

---

app/templates/edit_profile.html: Profile editor form

```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 %}
```

And finally, here is the view function that ties everything together:

app/routes.py: Edit profile view function

```
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)
```

This view function processes the form in a slightly different way. If `validate_on_submit()` returns `True` I copy the data from the form into the user object and then write the object to the database. But when `validate_on_submit()` returns `False` it can be due to two different reasons. First, it can be because the browser just sent a `GET` request, which I need to respond by providing an initial version of the form template. It can also be when the browser sends a `POST` request with form data, but something in that data is invalid. For this form, I need to treat these two cases separately. When the form is being requested for the first time with a `GET` request, I want to pre-populate the fields with the data that is stored in the database, so I need to do the reverse of what I did on the submission case and move the data stored in the user fields to the form, as this will ensure that those form fields have the current data stored for the user. But in the case of a validation error I do not want to write anything to the form fields, because those were already populated by WTForms. To distinguish between these two cases, I check

`request.method`, which will be `GET` for the initial request, and `POST` for a submission that failed validation.
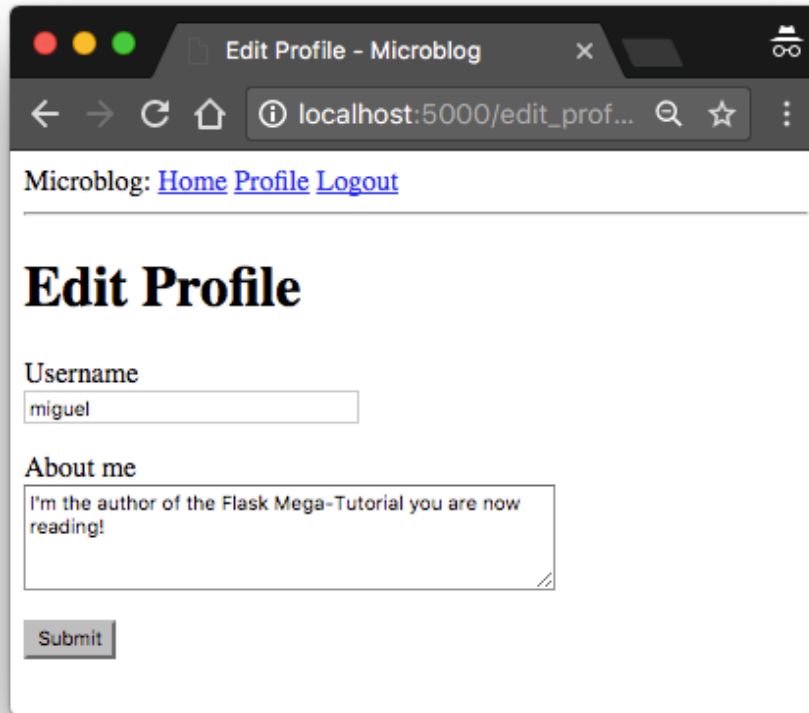
To make it easy for users to access the profile editor page, I can add a link in their profile page:

app/templates/user.html: Edit profile link

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

Pay attention to the clever conditional I'm using to make sure that the Edit link appears when you are viewing your own profile, but not when you are viewing the profile of someone else.

Continue on to the [next chapter](#).

Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through [Buy me a coffee](#). Thanks!

Share this post

Hacker News   Reddit   Twitter   LinkedIn

Facebook   E-Mail

30 comments

#1   **Vladimir Kuzmenkov**   said a year ago

The design is so intricate already, but feeling excited to implement all the powerful features.

#2 **Vyp** said a year ago

Hey Miguel, thank you for the wonderful tutorial!

I got stuck with the following problem in migrate and upgrade of database:
I made a typo in the User model, and generated a migration using
flask db migrate
Then I tried to upgrade the db
flask db upgrade
that obviously gave an error.

But now I fixed my typo in the User model and tried to run
flask db migrate
but that gives an error saying Target database is not up to date.
Of course
flask db upgrade
is still producing an error (because it is working with the old migration that had the typo)

So I am stuck between this cycle of errors between migrate and upgrade.

I can perhaps solve this by clearing all migrations, db, and re-initializing all that, but wondering if there is a better solution.

Thanks!

PS In case it matters, my typo in the first place that I missed the Optional in type specifier to about_me and last_seen columns

#3 Miguel Grinberg said a year ago

@Vyp: did you delete the bad migration file that was generated when you had the typo in the code?

#4 **Tree** said a year ago

if form.validate_on_submit():
TypeError: validate_on_submit() missing 1 required positional argument: 'self'

I have this error for calling validate_on_submit() when integrating EditProfileForm in the route? Any ideas sir? Thanks for your help.

#5 Miguel Grinberg said a year ago

@Tree: I publish the full working code on GitHub. Have you compared your code against mine? This is likely a minor error on your part in how you initialize your form class.

#6 **Giuseppe** said a year ago

Hey Miguel, I thank you for your very clear Tutorial and your time to make it.
I'm following your 2018 edition and I use Python3.7 as well as Thonny on a Raspberry pizero W. I'm a beginner.
Everything worked fine up to the end of the "Avatars" Section.
I created the Database repository,
I replied all the examples of the "Pay Time" Section (using the "flask shell"),
I successfully added the avatars to the HTML page.
However, when I attempted to generate the migration script it did'nt work giving the following error ..... venv/lib/python3.7/site-packages/email_validator/validate_email.py, line 10
/, # prior argumets are positional-only".
Exactly the same syntax error is now returned when I run the command "flask shell" under the <venv> .
I really can't figure out what caused the error.
And so, I hope you can suggest me a way to solve the problem.
Many thanks
Giuseppe

#7 Miguel Grinberg said a year ago

@Giuseppe: When you provide an error, you have to include to complete stack trace. A partial stack trace does not contain enough context.

#8 **Gery** said a year ago

Hi Miguel. I know its a stupid question, but I literally copied and pasted your full working code for this chapter, and when I did the flask run, it was like everything didn't show up as we expected. The avatar is missing, and even the profile link is missing, as is the Last_seen_on object. Since the code is perfect, I assumed there should be something wrong with the database. Do you have any clues about this? Thank you for your time and patience.

#9 **Miguel Grinberg** said a year ago

@Gery: the fully working code is available to you. Have you tried using this, or comparing it against your version? See the download link at the top of this article.

#10 **Gery** said a year ago

Yes I do, I downloaded the fully working code and open it in vs code, but those Avatar objects are still missing, so I think it could be something wrong with database stuff.

#11 **Miguel Grinberg** said a year ago

@Gary: you can always delete the database file and start over if you feel your database might be causing problems. My guess is that this is a code or template problem, not database. It's really hard to suggest anything else because I do not know exactly what you are seeing, maybe post a screenshot somewhere I can see it.

#12 **giuseppe** said a year ago

Hi Miguel,
as I told you in the previous post, following your suggestions now "localhost" works on the development machine.
At moment I'm at the end of Chapter 6. Up to the "Profile Editor" Section everything works fine. When I insert the code of the last section an error occurs:
" 'EditProfileForm' has no attribute 'about_me' ".

I checked the full code against the one available on GitHub without finding any differences. I attach the error I got. Thank you for all the help you're giving me.

(venv) raspberrypi-2@raspberrypi-2:~/Flask-related/FLASK-SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog6 $ flask run
* Serving Flask app 'main2023.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [12/Mar/2024 13:54:47] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Mar/2024 13:54:53] "GET /logout HTTP/1.1" 302 -
127.0.0.1 - - [12/Mar/2024 13:54:53] "GET /index HTTP/1.1" 302 -
127.0.0.1 - - [12/Mar/2024 13:54:53] "GET /login?next=/index HTTP/1.1" 200 -
127.0.0.1 - - [12/Mar/2024 13:55:03] "POST /login?next=/index HTTP/1.1" 302 -
127.0.0.1 - - [12/Mar/2024 13:55:03] "GET /index HTTP/1.1" 200 -
127.0.0.1 - - [12/Mar/2024 13:55:07] "GET /user/susan HTTP/1.1" 200 -
[2024-03-12 13:55:35,880] ERROR in app: Exception on /edit_profile [GET]
Traceback (most recent call last):
File "/home/raspberrypi-2/Flask-related/FLASK-SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog/venv/lib/python3.9/site-packages/flask/app.py", line 1463, in wsgi_app
response = self.full_dispatch_request()
File "/home/raspberrypi-2/Flask-related/FLASK-SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog/venv/lib/python3.9/site-packages/flask/app.py", line 872, in full_dispatch_request
rv = self.handle_user_exception(e)
File "/home/raspberrypi-2/Flask-related/FLASK-SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog/venv/lib/python3.9/site-packages/flask/app.py", line 870, in full_dispatch_request
rv = self.dispatch_request()
File "/home/raspberrypi-2/Flask-related/FLASK-

SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog/venv/lib/python3.9/site-packages/flask/app.py", line 855, in dispatch_request
return self.ensure_sync(self.view_functions[rule.endpoint])
(view_args) # type: ignore[no-any-return]
File "/home/raspberrypi-2/Flask-related/FLASK-SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog/venv/lib/python3.9/site-packages/flask_login/utils.py", line 290, in decorated_view
return current_app.ensure_sync(func)(*args, kwargs)
File "/home/raspberrypi-2/Flask-related/FLASK-SITO_WEB/Prove_2_Mega_Tutorial_2023/microblog6/appDir/routes.py", line 107, in edit_profile
form.about_me.data = current_user.about_me
AttributeError: 'EditProfileForm' object has no attribute 'about_me'
127.0.0.1 - - [12/Mar/2024 13:55:35] "GET /edit_profile HTTP/1.1" 500 -

#13   Miguel Grinberg   said a year ago

@giuseppe: the error message tells you exactly what's wrong:

```
AttributeError: 'EditProfileForm' object has no attri
```

◀ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ ▶

Look at the definition of the `EditProfileForm` class. It should have a property called `about_me`. This property is shown above in this article. The error indicates that you don't have it in your version of the code. Either because you missed it completely, or because you typed a different name for it.

#14   **Sergey**   said a year ago

Hello, Migel!
Thank you for the perfect learning material! I ran into a little problem. On the profile editing form, if you enter an existing user name in the "User" field, an unhandled exception occurs:
«sqlalchemy.exc.IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed: user.username
[SQL: UPDATE user SET username=?, about_me=? WHERE user.id = ?]

[parameters: ('susan', '123456', 2)]»
Is there a way to solve the problem?

#15 Miguel Grinberg said a year ago

@Sergey: This is addressed later in the tutorial.

#16 **Shaun** said a year ago

Hi Miguel,

Thanks for an awesome tutorial.

I've an issue with the commit step not updating a field on an object.

The project is currently (stripped down):
- app
-- **init**.py
-- models.py
-- routes.py

It follows the tutorial mostly, however, I have another class called 'script':

```
class Script(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=
    name: so.Mapped[str] = so.mapped_column(sa.String(
                                            unique
    script: so.Mapped[str] = so.mapped_column(sa.Strin
                                            unique
    day: so.Mapped[str] = so.mapped_column(sa.String(1
                                            unique=Fa
    time: so.Mapped[str] = so.mapped_column(sa.String(
                                            unique=Fa
    last_update: so.Mapped[Optional[DateTime]] = so.ma
                                            unique=Fa


    def __repr__(self):
        return '<Script {}, at {}>'.format(self.name,
```

◄   _____   ►

It is imported in **init**.py:

```
from flask import Flask
...
from app import routes, models
```

and again in routes.py:
…
from app.models import Script

But when I try to update the script it doesn't update the DB from within the running application. However, the update works when updating manually via 'flask shell'

Any pointers?

#17   **Miguel Grinberg**   said a year ago

@Shaun: I don't understand this:

> *But when I try to update the script it doesn't update the DB from within the running application. However, the update works when updating manually via 'flask shell'*

Can you please show me exactly what you are doing in the working and non-working cases?

#18   **Mindaugas**   said a year ago

Hi Miguel, thanks for the tutorial. Hope flask is still relevant nowadays! I'm curious if using database operations in app.before_request without conditionals checks generally isn't a bad practice from performance perspective? Why not just update last seen on user profile page visit?

#19   **Miguel Grinberg**   said a year ago

@Mindaugas: the point of having a last seen date is for it to record the last time the user accessed the application, regardless of which page it was. Doing it in the before_request handler ensures this. If you find that this additional database

activity affects performance then you can just no do it, or only record the user's login date and time.

#20   **Doniphan Pattison**   said 10 months ago

It's not that hard to move identicons onto flask instance...

```
pip install pydenticon
```

models:

```
import pydenticon, hashlib, base64
...
class User...
    def gen_avatar(self, write_png=True):
        foreground = [ "rgb(45,79,255)",
                  "rgb(254,180,44)",
                  "rgb(226,121,234)",
                  "rgb(30,179,253)",
                  "rgb(232,77,65)",
                  "rgb(49,203,115)",
                  "rgb(141,69,170)" ]
        background = "rgb(22,22,22)"

        digest = hashlib.md5(self.email.lower().encode
        basedir = os.path.abspath(os.path.dirname(__fi
        pngloc = os.path.join(basedir, 'usercontent',
        icongen = pydenticon.Generator(5, 5, digest=ha
        pngicon = icongen.generate(self.email, 120, 12
        if write_png:
            pngfile = open(pngloc, "wb")
            pngfile.write(pngicon)
            pngfile.close()
        else:
            return str(base64.b64encode(pngicon))[2:-1
```

usertemplate:

```
<img src="data:image/png;base64,{{ user.gen_avatar(wr:
```

#21   **anna**   said 9 months ago

hey Miguel! I have been loving this tutorial so far! very comprehensible and its turning out great :)
I'm actually designing my own project with flask. My plan is to alter the blog I create following your tutorial, and transform it into the website I need. However, I looked over some chapters here, and it seems like some of them are going to be unnecessary for me. Would I be able to follow the tutorial if I skip 2-3 chapters, or would your recommend to go through all of the chapters, since within them could be some vital components?

#22   Miguel Grinberg   said 9 months ago

@anna: You can skip chapters that aren't interesting to you, and then you can come back to them later if you find that something you are interested in requires undestanding those parts that you missed.

#23   **Nate Jordan**   said 7 months ago

Many thanks Miguel! You are doing an amazing service to the Python/Flask students of the world!

My question/observation: everything works up to this point. I just notice that in the terminal, while clicking through the app, when I click the profile page I get these 2 lines:

> *127.0.0.1 - - [11/Oct/2024 06:19:46] "GET /user/nate HTTP/1.1" 200 -*
> *127.0.0.1 - - [11/Oct/2024 06:19:46] "GET /user/None HTTP/1.1" 404 -*

Is that normal? Again, everything seems to be working fine. Just curious. :)

Thank you!

#24   Miguel Grinberg   said 7 months ago

@Nate: only the first is normal, the second one is not. You may want to compare your code against mine to see if you spot any differences.

Laza said 7 months ago

I was getting an error. The error was stating that the column, about-me, has to be nullable or create a default value. Nothing I was doing was working, finally I decided to go and look in the migration script, as shown below:

```
def upgrade():
# ### commands auto generated by Alembic - please adjust!
###
################################################
################
batch_op.add_column(sa.Column('about_me',
sa.String(length=140), nullable=True))
################################################
#############
```

I had to change the nullable from 'False' to 'True' here. It was not working the other way around, my guess is that when the file runs for the first time, it gets cached or something..

But it works now.. awesome tuts.

Leave a Comment

Name

Email

Comment

Captcha

Submit

Flask Web Development, 2nd Edition

If you want to learn modern web development techniques with Python and Flask, you may find the second edition of my O'Reilly book useful.

Click here to get this Book!

About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.

You can also find me on Github, LinkedIn, Bluesky, Mastodon, Twitter, YouTube, and Patreon.

Thank you for visiting!

Categories

- AI *2*
- Arduino *7*
- Authentication *10*
- Blog *1*
- C++ *5*
- CSS *1*
- Cloud *11*
- Database *23*
- Docker *5*
- Filmmaking *6*
- Flask *128*
- Games *1*
- IoT *8*
- JavaScript *35*
- MicroPython *9*
- Microdot *1*
- Microservices *2*
- Movie Reviews *5*
- Personal *3*
- Photography *7*
- Product Reviews *2*
- Programming *191*
- Project Management *1*
- Python *173*
- REST *7*
- Raspberry Pi *8*
- React *18*
- Reviews *1*

🔶 Robotics *6*

🔶 Security *12*

🔶 Video *22*

🔶 WebSocket *2*

🔶 Webcast *3*

🔶 Windows *1*