Allowing users to login to your app is one of the most common features you'll add to a web app you build. This article will cover how to add simple authentication to your Flask app. The main package we will use to accomplish this is Flask Login.

Table of Contents

What We'll Be Building

We're going to build some sign up and login pages that allows our app to allow users to login and access protected pages that non-logged in users can see. We'll grab information from the user model and display it on our protected pages when the user logs in to simulate what a profile would look like.

We will cover the following in this article:

- Use the Flask-Login library for session management
- Use the built-in Flask utility for hashing passwords
- Add protected pages to our app for logged in users only
- Use Flask-SQLAlchemy to create a user model
- Create sign up and login forms for our users to create accounts and login
- Flash error messages back to users when something goes wrong
- Use information from the user's account to display on the profile page

Setting Up The Application

Our app will use the Flask app factory pattern with blueprints. We'll have one blueprint that handles everything auth related, and we'll have another blueprint for our regular routes, which include the index and the protected profile page. In a real app, of course, you can break down the functionality in any way you like, but what I've proposed will work well for this tutorial.

To start, we need to create the directories and files for our project.

```
- project
---- templates
-------- base.html <!-- contains common layout and
-------- index.html <!-- show the home page -->
-------- login.html <!-- show the login form -->
-------- profile.html <!-- show the profile page -
-------- signup.html <!-- show the signup form -->
---- __init__.py <!-- setup our app -->
---- auth.py <!-- the auth routes for our app -->
---- main.py <!-- the non-auth routes for our app
---- models.py <!-- our user model -->
```

You can create those files and we'll add them as we progress along.

Install Packages

There are three main packages we need for our project:

- Flask
- Flask-Login - to handle the user sessions after authentication
- Flask-SQLAlchemy - to represent the user model and interface with our database

We'll only be using SQLite for the database to avoid having to install any extra dependencies for the database. Here's what

you need to run after creating your virtual environment to install the packages.

```
pip install flask flask-sqlalchemy flask-login
```

Main App File

Let's start by creating the __init__.py file for our project. This will have the function to create our app which will initialize the database and register our blueprints. At the moment this won't do much, but it will be needed for the rest of our app. All we need to do is initialize SQLAlchemy, set some configuration values, and register our blueprints here.

__init__.py

```
# __init__.py

from flask import Flask__
from flask_sqlalchemy import SQLAlchemy

# init SQLAlchemy so we can use it later in our mo
db = SQLAlchemy()

def create_app():
    app = Flask(__name__)
```

```
    app.config['SECRET_KEY'] = '9OLWxND4o83j4K4iuo
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlit

    db.init_app(app)

    # blueprint for auth routes in our app
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint)

    # blueprint for non-auth parts of app
    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

Route Scaffolding

Now that we have the main app file, we can start adding in our routes.

For our routes, we'll use two blueprints. For our main blueprint, we'll have a home page (/) and profile page (/profile) for after we log in. If the user tries to access the profile page without being logged in, they'll be sent to our login route.

For our auth blueprint, we'll have routes to retrieve both the login page (/login) and signup page (/signup). We'll also have routes for handling the POST request from both of those two routes. Finally, we'll have

a logout route (/logout) to logout an active user.

Let's go ahead and add them even though they won't do much. Later we will update them so we can use them.

main.py

```
# main.py

from flask import Blueprint
from . import db

main = Blueprint('main', __name__)

@main.route('/')
def index():
    return 'Index'

@main.route('/profile')
def profile():
    return 'Profile'
```

auth.py

```
# auth.py

from flask import Blueprint
```

```
from . import db

auth = Blueprint('auth', __name__)

@auth.route('/login')
def login():
    return 'Login'

@auth.route('/signup')
def signup():
    return 'Signup'

@auth.route('/logout')
def logout():
    return 'Logout'
```

You can now set the FLASK_APP and FLASK_DEBUG values and run the project. You should be able to view navigate to the five possible URLs and see the text returned.

```
export FLASK_APP=project
export FLASK_DEBUG=1
flask run
```

Templates

Let's go ahead and create the templates that are used in our app. This is the first step before we can implement the actual login functionality. Our app will use four templates:

- index.html
- profile.html
- login.html
- signup.html

We'll also have a base template that will have code common to each of the pages. In this case, the base template will have navigation links and the general layout of the page. Let's create them now.

templates/base.html

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

<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE
    <meta name="viewport" content="width=device-wi
    <title>Flask Auth Example</title>
    <link rel="stylesheet" href="https://cdnjs.clo
</head>

<body>
    <section class="hero is-primary is-fullheight"

        <div class="hero-head">
            <nav class="navbar">
                <div class="container">

                    <div id="navbarMenuHeroA" clas
                        <div class="navbar-end">
                            <a href="{{ url_for('m
                                Home
                            </a>
                            <a href="{{ url_for('m
                                Profile
                            </a>
                            <a href="{{ url_for('a
                                Login
                            </a>
                            <a href="{{ url_for('a
```

```
                                Sign Up
                            </a>
                            <a href="{{ url_for('a
                                Logout
                            </a>
                        </div>
                    </div>
                </div>
            </nav>
        </div>

        <div class="hero-body">
            <div class="container has-text-centere
               {% block content %}
               {% endblock %}
            </div>
        </div>
    </section>
</body>

</html>
```

templates/index.html

```
<!-- templates/index.html -->

{% extends "base.html" %}

{% block content %}
<h1 class="title">
```

```
    Flask Login Example
  </h1>
  <h2 class="subtitle">
    Easy authentication and authorization in Flask.
  </h2>
  {% endblock %}
```

templates/login.html

```
<!-- templates/login.html -->

{% extends "base.html" %}

{% block content %}
<div class="column is-4 is-offset-4">
    <h3 class="title">Login</h3>
    <div class="box">
        <form method="POST" action="/login">
            <div class="field">
                <div class="control">
                    <input class="input is-large"
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large"
                </div>
            </div>
            <div class="field">
```

```
                <label class="checkbox">
                    <input type="checkbox">
                    Remember me
                </label>
            </div>
            <button class="button is-block is-info
        </form>
    </div>
</div>
{% endblock %}
```

templates/signup.html

```
<!-- templates/signup.html -->

{% extends "base.html" %}

{% block content %}
<div class="column is-4 is-offset-4">
    <h3 class="title">Sign Up</h3>
    <div class="box">
        <form method="POST" action="/signup">
            <div class="field">
                <div class="control">
                    <input class="input is-large"
                </div>
            </div>

            <div class="field">
                <div class="control">
```

```
                    <input class="input is-large"
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large"
                </div>
            </div>

            <button class="button is-block is-info
        </form>
    </div>
</div>
{% endblock %}
```

templates/profile.html

```
<!-- templates/profile.html -->

{% extends "base.html" %}

{% block content %}
<h1 class="title">
  Welcome, Anthony!
</h1>
{% endblock %}
```

Once you've added the templates, we can update the return statements in each of the routes we have to return the templates instead of the text.

main.py

```
# main.py

from flask import Blueprint, render_template
...
@main.route('/')
def index():
    return render_template('index.html')

@main.route('/profile')
def profile():
    return render_template('profile.html')
```

main.py

```
# auth.py

from flask import Blueprint, render_template
...
@auth.route('/login')
def login():
    return render_template('login.html')
```

```
@auth.route('/signup')
def signup():
    return render_template('signup.html')
```

For example, here is what the signup page looks like if you navigate to /signup. You should be able see the pages for /, /login, and /profile as well. We'll leave /logout alone for now because it won't display a template when it's done.

User Model

Our user model represents what it means for our app to have a user. To keep it simple, we'll have fields for an email address, password, and name. Of course in your application, you may decide you want much more information to be stored per user. You can add things like birthday, profile picture, location, or any user preferences.

Models created in Flask-SQLAlchemy are represented by classes which then translate

to tables in a database. The attributes of those classes then turn into columns for those tables.

Let's go ahead and create that user model.

models.py

```
# models.py

from . import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True) #
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(1000))
```

Database Config

Like I said before, we'll be using a SQLite database. We could create a SQLite database on our own, but let's have Flask-SQLAlchemy do it for us.

We already have the path of the database specified in the __init__.py file, so we just need to tell Flask-SQLAlchemy to create the database for us in the Python REPL.

If you stop your app and open up a Python REPL, we can create the database using the create_all method on the db object.

```
from project import db, create_app

db.create_all(app=create_app()) # pass the create_
```

You should now see a db.sqlite file in your project directory. This database will have our user table in it.

Sign up Method

Now that we have everything set up, we can finally get to writing the code for the authorization.

For our sign up function, we're going to take the data the user types into the form and add it to our database. But before we add it, we need to make sure the user

doesn't already exist in the database. If it doesn't, then we need to make sure we hash the password before placing it into the database, because we don't want our passwords stored in plaintext.

Let's start by adding a second function to handle the POSTed form data. In this function, we will gather the data passed from the user first.

Let's start by creating the function and adding a redirect to the bottom because we know when we add the user to the database, we will redirect to the login route.

auth.py

```
# auth.py

from flask import Blueprint, render_template, redi
...
@auth.route('/signup', methods=['POST'])
def signup_post():
    # code to validate and add user to database go
    return redirect(url_for('auth.login'))
```

Now, let's add the rest of the code necessary for signing up a user.

To start, we'll have to use the request object to get the form data. If you're not familar with the request object, I wrote on article

on it here: https://scotch.io/bar-talk/processing-incoming-request-data-in-flask

auth.py

```
# auth.py

from flask import Blueprint, render_template, redi
from werkzeug.security import generate_password_ha
from .models import User
from . import db
...
@auth.route('/signup', methods=['POST'])
def signup_post():
    email = request.form.get('email')
    name = request.form.get('name')
    password = request.form.get('password')

    user = User.query.filter_by(email=email).first

    if user: # if a user is found, we want to redi
        return redirect(url_for('auth.signup'))

    # create new user with the form data. Hash the
    new_user = User(email=email, name=name, passwo

    # add the new user to the database
    db.session.add(new_user)
    db.session.commit()

    return redirect(url_for('auth.login'))
```

Test Sign Up Method

Now that we have the sign up method done, we should be able to create a new user. Use the form to create a user.

There are two ways you can verify if the sign up worked: you can use a database viewer to look at the row that was added to your table, or you can simply try signing up with the same email address again, and if you get an error, you know the first email was saved properly. So let's take that approach.

We can add code to let the user know the email already exists and tell them to go to the login page. By calling the flash function, we will send a message to the next request, which in this case, is the redirect. The page we land on will then have access to that message in the template.

First, we add the flash before we redirect back to our signup page.

auth.py

```
# auth.py

from flask import Blueprint, render_template, redi
```

```
...
@auth.route('/signup', methods=['POST'])
def signup_post():
    ...
    if user: # if a user is found, we want to redi
        flash('Email address already exists')
        return redirect(url_for('auth.signup'))
```

To get the flashed message in the template, we can add this code above the form. This will display the message directly above the form.

templates/signup.html

```
<!-- templates/signup.html -->
...
{% with messages = get_flashed_messages() %}
{% if messages %}
    <div class="notification is-danger">
        {{ messages[0] }}. Go to <a href="{{ url_f
    </div>
{% endif %}
{% endwith %}
<form method="POST" action="/signup">
...
```

Login Method

The login method is similiar to the signup function in that we will take the user information and do something with it. In this case, we will compare the email address entered to see if it's in the database. If so, we will test the password the user provided by hashing the password the user passes in and comparing it to the hashed password in the database. We know the user has entered the correct password when both hashed passwords match.

Once the user has passed the password check, we know that they have the correct credentials and we can go ahead and log them in using Flask-Login. By calling login_user, Flask-Login will create a session for that user that will persist as the user stays logged in, which will allow the user to view protected pages.

We can start with a new route for handling the POSTed data. We'll redirect to the profile page when the user successfully logs in.

auth.py

```
# auth.py
...
@auth.route('/login', methods=['POST'])
def login_post():
    # login code goes here
    return redirect(url_for('main.profile'))
```

Now, we need to verify if the user has the correct credentials.

auth.py

```
# auth.py
...
@auth.route('/login', methods=['POST'])
def login_post():
    email = request.form.get('email')
    password = request.form.get('password')
    remember = True if request.form.get('remember'

    user = User.query.filter_by(email=email).first

    # check if user actually exists
    # take the user supplied password, hash it, an
    if not user or not check_password_hash(user.pa
        flash('Please check your login details and
```

```
        return redirect(url_for('auth.login')) # i

    # if the above check passes, then we know the

    return redirect(url_for('main.profile'))
```

Let's add in the block in the template so the user can see the flashed message. Like the signup form, let's add the potential error message directly above the form.

templates/login.html

```
<!-- templates/login.html -->
...
{% with messages = get_flashed_messages() %}
{% if messages %}
    <div class="notification is-danger">
        {{ messages[0] }}
    </div>
{% endif %}
{% endwith %}
<form method="POST" action="/login">
```

So we have have the ability to say a user has been logged in succesfully, but there is nothing to actually log the user in anywhere. This is where we bring in Flask-Login.

But first, we need a few things for Flask-Login to work.

We start by adding something called the UserMixin to our User model. The UserMixin will add Flask-Login attributes to our model so Flask-Login will be able to work with it.

models.py

```
# models.py

from flask_login import UserMixin
from . import db

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True) #
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(1000))
```

Then, we need to specify our user loader. A user loader tells Flask-Login how to find a specific user from the ID that is stored in their session cookie. We can add this in our create_app function along with basic init code for Flask-Login.

__init__.py

```
# __init__.py

...
from flask_login import LoginManager

def create_app():
    ...
    db.init_app(app)

    login_manager = LoginManager()
    login_manager.login_view = 'auth.login'
    login_manager.init_app(app)

    from .models import User

    @login_manager.user_loader
    def load_user(user_id):
        # since the user_id is just the primary ke
        return User.query.get(int(user_id))
```

Finally, we can add the login_user function just before we redirect to the profile page to create the session.

auth.py

```
# auth.py
```

```
from flask_login import login_user
from .models import User
...
@auth.route('/login', methods=['POST'])
def login_post():
    # if the above check passes, then we know the
    login_user(user, remember=remember)
    return redirect(url_for('main.profile'))
```

With Flask-Login setup, we can finally use the /login route.

When everything is successful, we will see the profile page.

Protected Pages

If your name isn't also Anthony, then you'll see that your name is wrong. What we want is the profile to display the name in the database. So first, we need to protect the

page and then access the user's data to get the name.

To protect a page when using Flask-Login is very simple: we add the @login_requried decorator between the route and the function. This will prevent a user who isn't logged in from seeing the route. If the user isn't logged in, the user will get redirected to the login page, per the Flask-Login configuration.

With routes that are decorated with the login_required decorator, we then have the ability to use the current_user object inside of the function. This current_user represents the user from the database, and we can access all of the attributes of that user with dot notation. For example, current_user.email, current_user.password, and current_user.name and current_user.id will return the actual values stored in the database for the logged in user.

Let's use the name of the current user and send it to the template. We then will use that name and display its value.

main.py

```
# main.py

from flask import Blueprint, render_template
from flask_login import login_required, current_us
...
@main.route('/profile')
```

```
@login_required
def profile():
    return render_template('profile.html', name=cu
```

templates/profile.html

```
<!-- templates/profile.html -->
...
<h1 class="title">
  Welcome, {{ name }}!
</h1>
```

Once we go to our profile page, we then see that the user's name appears.

The final thing we can do is update our logout view. We can call the logout_user function in a route for logging out. We have the login_required decorator because it doesn't make sense to logout a user who isn't logged in to begin with.

auth.py

```
# auth.py

from flask_login import login_user, logout_user, l
...
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('main.index'))
```

After we logout and try viewing the profile page again, we see a error message appear. This is because Flask-Login flashes a message for us when the user isn't allowed to access a page.

One last thing we can do is put if statements in the templates to display only the links relevant to the user. So before the user logins in, they will have the option to

login or signup. After they have logged in, they can go to their profile or logout.

templates/base.html

```
<!-- templates/base.html -->
...
<div class="navbar-end">
    <a href="{{ url_for('main.index') }}" class="n
        Home
    </a>
    {% if current_user.is_authenticated %}
    <a href="{{ url_for('main.profile') }}" class=
        Profile
    </a>
    {% endif %}
    {% if not current_user.is_authenticated %}
    <a href="{{ url_for('auth.login') }}" class="n
        Login
    </a>
    <a href="{{ url_for('auth.signup') }}" class="
        Sign Up
    </a>
    {% endif %}
    {% if current_user.is_authenticated %}
    <a href="{{ url_for('auth.logout') }}" class="
        Logout
    </a>
    {% endif %}
</div>
```

Conclusion

We've done it! We have used Flask-Login and Flask-SQLAlchemy to build a very basic login system for our app. We covered how to authenticate a user by first creating a user model and storing the user information to later. Then we had to verify the user's password was correct by hashing the password from the form and comparing it to the one stored in the database. Finally, we added authorization to our app by using the @login_required decotor on a profile page so only logged in users can see that page.

What we created in this tutorial will be sufficient for smaller apps, but if you wish to have more functionality from the beginning, you may want to consider using either the Flask-User or Flask-Security libraries, which are both build on top of the Flask-Login library.

Like this article? Follow @pretty_printed on Twitte