undefined Logo

Handle User Accounts & Authentication in Flask with Flask-Login

Todd Flask 15 Min Read

We've covered a lot of Flask goodness in this series thus far. We fully understand how to structure a sensible application; we can serve up complex page templates with Jinja, and

we've tackled interacting with databases using Flask-SQLAlchemy. For our next challenge, we're going to need *all* the knowledge we've acquired thus far and much, much more. Welcome to the Super Bowl of Flask development. This. Is. **Flask-Login.**

Flask-Login is a dope library that handles all aspects of user management, including user signups, encrypting passwords, managing sessions, and securing parts of our app behind login walls. Flask-Login also happens to play nicely with other Flask libraries we're already familiar with! There's built-in support for storing user data in a database via **Flask-SQLAlchemy**, while **Flask-WTForms** covers the subtleties of signup & login form logic. This tutorial assumes you have *some* working knowledge of these libraries.

Flask-Login is shockingly quite easy to use after the initial learning curve, but therein lies the rub. Perhaps I'm not the only one to have noticed, but most Flask-related documentation tends to be, well, God-awful. The community is riddled with helplessly outdated information; if you ever come across import flask.ext.plugin_name in a tutorial, it is inherently worthless to anybody developing in 2019. To make matters worse, official Flask-Login documentation contains some artifacts which are just plain wrong. The documentation contradicts itself (I'll show you what I mean), and offers little to no code examples to speak of. My only hope is that I might save somebody the endless headaches I've experienced myself.

For reference, a live demo of this tutorial is available here: https://flasklogin.hackersandslackers.app/

Getting Started

Let's start with installing dependencies. Depending on which flavor of database you prefer, install either `psycopg2-binary` or `pymysql` along with the following:

Install dependencies

```
$ pip3 install flask flask-login flask-sqlalchemy flask-wtf
```

It should be clear as to why we need each of these packages. Handling user accounts requires a database to store user data, hence `flask-sqlalchemy`. We'll be capturing that data via forms using `flask-wtf`.

I'm going to store all routes related to user authentication under **auth.py**, and the rest of our "logged-in" routes in **routes.py**. The rest should be clear as per the standard procedure:

Project structure.

```
/flasklogin-tutorial
├── /flask_login_tutorial
│   ├── __init__.py
│   ├── assets.py
│   ├── auth.py
│   ├── forms.py
│   ├── models.py
│   ├── routes.py
│   ├── /static
│   └── /templates
├── config.py
├── requirements.txt
├── start.sh
└── wsgi.py
```

Configuring for Flask-Login

I wouldn't be a gentleman unless I revealed my **config.py** file. Again, this should all be straightforward if you've been following the rest of our series:

config.py

```
"""Flask app configuration."""
from os import environ, path
from dotenv import load_dotenv

basedir = path.abspath(path.dirname(__file__))
load_dotenv(path.join(basedir, '.env'))


class Config:
    """Set Flask configuration from environment variables."""

    FLASK_APP = 'wsgi.py'
    FLASK_ENV = environ.get('FLASK_ENV')
    SECRET_KEY = environ.get('SECRET_KEY')

    # Flask-Assets
    LESS_BIN = environ.get('LESS_BIN')
    ASSETS_DEBUG = environ.get('ASSETS_DEBUG')
    LESS_RUN_IN_DEBUG = environ.get('LESS_RUN_IN_DEBUG')

    # Static Assets
    STATIC_FOLDER = 'static'
    TEMPLATES_FOLDER = 'templates'
    COMPRESSOR_DEBUG = environ.get('COMPRESSOR_DEBUG')

    # Flask-SQLAlchemy
    SQLALCHEMY_DATABASE_URI = environ.get('SQLALCHEMY_DATABASE_URI')
```

```
    SQLALCHEMY_ECHO = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Of these variables, SECRET_KEY and SQLALCHEMY_DATABASE_URI deserve our attention. SQLALCHEMY_DATABASE_URI is a given, as this is how we'll be connecting to our database.

Flask's SECRET_KEY variable is a string used to encrypt all of our user's passwords (or other sensitive information). We should strive to set this string to be as long, nonsensical, and impossible-to-remember as humanly possible. **Protect this string with your life**: anybody who gets their hands on your app's secret key has the power to potentially *unencrypted* all user passwords in your application. In an ideal world, even *you* shouldn't know your key. Seriously: having your secret key compromised is the equivalent of feeding gremlins after midnight.

Initializing Flask-Login

Setting up Flask-Login via the application factory pattern is no different from using any other Flask plugin (or whatever they're called now). This makes setting up easy: all we need to do is make sure **Flask-Login** is initialized in `__init__.py` along with the rest of our plugins, as we do with **Flask-SQLAlchemy**:

__ init __.py

```
"""Initialize app."""
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager


db = SQLAlchemy()
login_manager = LoginManager()


def create_app():
    """Construct the core app object."""
    app = Flask(__name__, instance_relative_config=False)

    # Application Configuration
    app.config.from_object('config.Config')

    # Initialize Plugins
    db.init_app(app)
    login_manager.init_app(app)

    with app.app_context():
        from . import routes
```

```
        from . import auth
        from .assets import compile_assets

        # Register Blueprints
        app.register_blueprint(routes.main_bp)
        app.register_blueprint(auth.auth_bp)

        # Create Database Models
        db.create_all()

        # Compile static assets
        if app.config['FLASK_ENV'] == 'development':
            compile_assets(app)

        return app
```

This is the minimum we need to set up Flask-Login properly.

As mentioned earlier, we're separating routes pertaining to user authentication from our main application routes. We handle this by registering two blueprints: **auth_bp** is imported from **auth.py**, and our "main" application routes are associated with **main_bp** from **routes.py**. We'll be digging into both of these blueprints, but let's first turn our focus to preparing our database to store users.

Creating a User Model

Flask-Login is tightly coupled with Flask-SQLAlchemy's ORM, which makes creating and validating users trivially easy. All we need to do upfront is to create a **User** model, which we'll keep in **models.py**. It helps to be familiar with the basics creating database models in Flask here; if you've skipped ahead, well... that's your problem.

The most notable tidbit for creating a User model is a nifty shortcut from Flask-Login called the **UserMixin**. **UserMixin** is a helper provided by the Flask-Login library to provide boilerplate methods necessary for managing users. Models which inherit UserMixin immediately gain access to 4 useful methods:

- `is_authenticated`: Checks to see if the current user is already authenticated, thus allowing them to bypass login screens.
- `is_active`: If your app supports disabling or temporarily banning accounts, we can check `if user.is_active()` to handle a case where their account exists, but have been banished from the land.
- `is_anonymous`: Many apps have a case where user accounts aren't entirely black-and-white, and anonymous users have access to interact without authenticating. This method might come in handy for allowing anonymous blog comments (which is madness, by the way).

- `get_id`: Fetches a unique ID identifying the user.

Creating a User model via **UserMixin** is by far the easiest way of getting started- the bulk of what remains is specifying the fields we want to capture for users. At a minimum, I'd suggest a username/email and password:

models.py

```
"""Database models."""
from . import db
from flask_login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash


class User(UserMixin, db.Model):
        """User account model."""

    __tablename__ = 'flasklogin-users'
    id = db.Column(
        db.Integer,
        primary_key=True
    )
    name = db.Column(
        db.String(100),
        nullable=False,
        unique=False
    )
    email = db.Column(
        db.String(40),
        unique=True,
        nullable=False
    )
    password = db.Column(
        db.String(200),
        primary_key=False,
        unique=False,
        nullable=False
        )
    website = db.Column(
        db.String(60),
        index=False,
        unique=False,
        nullable=True
        )
    created_on = db.Column(
        db.DateTime,
        index=False,
        unique=False,
        nullable=True
    )
    last_login = db.Column(
        db.DateTime,
```

```
        index=False,
        unique=False,
        nullable=True
    )

    def set_password(self, password):
        """Create hashed password."""
        self.password = generate_password_hash(
            password,
            method='sha256'
        )

    def check_password(self, password):
        """Check hashed password."""
        return check_password_hash(self.password, password)

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

You may notice that our password field explicitly allows 200 characters: this is because our database will be storing hashed passwords. Thus, even if a user's password is eight characters long, the string in our database will look much different.

It's nice to keep logic related to users bundled in our model and out of our routes, hence the `set_password()` and `check_password()` methods. These methods leverage the `werkzeug` library to handle the hashing and checking of passwords (this is what depends on our SECRET_KEY from earlier to store passwords securely).

Creating Log-in and Sign-up Forms

Hopefully you've become somewhat acquainted with Flask-WTF or WTForms in the past. We create two form classes in **forms.py** which cover our most essential needs: a sign-up form, and a log-in form:

forms.py

```
"""Sign-up & log-in forms."""
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, EqualTo, Length, Optional


class SignupForm(FlaskForm):
    """User Sign-up Form."""
    name = StringField(
        'Name',
        validators=[DataRequired()]
    )
```

```
    email = StringField(
        'Email',
        validators=[
            Length(min=6),
            Email(message='Enter a valid email.'),
            DataRequired()
        ]
    )
    password = PasswordField(
        'Password',
        validators=[
            DataRequired(),
            Length(min=6, message='Select a stronger password.')
        ]
    )
    confirm = PasswordField(
        'Confirm Your Password',
        validators=[
            DataRequired(),
            EqualTo('password', message='Passwords must match.')
        ]
    )
    website = StringField(
        'Website',
        validators=[Optional()]
    )
    submit = SubmitField('Register')


class LoginForm(FlaskForm):
    """User Log-in Form."""
    email = StringField(
        'Email',
        validators=[
            DataRequired(),
            Email(message='Enter a valid email.')
        ]
    )
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Log In')
```

Be sure to add fields in your form for whichever fields you specified in your **User** model, unless you're okay with having users change this information later. In my case, `created_on` and `last_login` are empty as these are easily handled on the SQL side of things.

Here is where we might expect to jump into creating routes for our signup and login pages. Instead of blowing your mind with the information overload that section is about to inflict on your unsuspecting mind, let's get comfortable by quickly looking at the Jinja templates for the signup and login forms we just created.

signup.jinja2

The signup form is the heavier of the two forms as there's far more validation associated with creating a brand new user account as opposed to validating an email and password. The form template below wraps each field of the signup form in a `<fieldset>` tag, where we keep our field, our field's label, and all error message handling associated with submitting an invalid form:

signup.jinja

```
{% extends "layout.jinja2" %}


{% block content %}
  <div class="form-wrapper">

    <div class="logo">
      <img src="{{ url_for('static', filename='dist/img/logo.png') }}" alt="logo"/>
    </div>

    {% for message in get_flashed_messages() %}
      <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&amp;times;</button>
        {{ message }}
      </div>
    {% endfor %}

    <h1>Sign Up</h1>

    <form method="POST" action="/signup">
      {{ form.csrf_token }}

      <fieldset class="name">
        {{ form.name.label }}
        {{ form.name(placeholder='John Smith') }}
        {% if form.name.errors %}
          <ul class="errors">
            {% for error in form.email.errors %}
              <li>{{ error }}</li>{% endfor %}
          </ul>
        {% endif %}
      </fieldset>

      <fieldset class="email">
        {{ form.email.label }}
        {{ form.email(placeholder='youremail@example.com') }}
        {% if form.email.errors %}
          <ul class="errors">
            {% for error in form.email.errors %}
              <li>{{ error }}</li>{% endfor %}
          </ul>
        {% endif %}
```

```
      </fieldset>

      <fieldset class="password">
        {{ form.password.label }}
        {{ form.password }}
        {% if form.password.errors %}
          <ul class="errors">
            {% for error in form.password.errors %}
              <li>{{ error }}</li>{% endfor %}
          </ul>
        {% endif %}
      </fieldset>

      <fieldset class="confirm">
        {{ form.confirm.label }}
        {{ form.confirm }}
        {% if form.confirm.errors %}
          <ul class="errors">
            {% for error in form.confirm.errors %}
              <li>{{ error }}</li>{% endfor %}
          </ul>
        {% endif %}
      </fieldset>

      <fieldset class="website">
        {{ form.website.label }}
        {{ form.website(placeholder='http://example.com') }}
      </fieldset>

      <div class="submit-button">
        {{ form.submit }}
      </div>

    </form>

    <div class="login-signup">
      <span>Already have an account? <a href="{{ url_for('auth_bp.login') }}">Log in.</a></span>
    </div>
  </div>
{% endblock %}
```

login.jinja2

The login form template contains essentially the same structure as our signup form, but with fewer fields:

login.jinja2

```
{% extends "layout.jinja2" %}
```

```
{% block content %}
  <div class="form-wrapper">

    <div class="logo">
      <img src="{{ url_for('static', filename='dist/img/logo.png') }}" alt="logo"/>
    </div>

    {% for message in get_flashed_messages() %}
      <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        {{ message }}
      </div>
    {% endfor %}

    <h1>Log In</h1>

    <form method="POST" action="/login">
      {{ form.csrf_token }}

      <fieldset class="email">
        {{ form.email.label }}
        {{ form.email(placeholder='youremail@example.com') }}
        {% if form.email.errors %}
          <ul class="errors">
            {% for error in form.email.errors %}
              <li>{{ error }}</li>{% endfor %}
          </ul>
        {% endif %}
      </fieldset>

      <fieldset class="password">
        {{ form.password.label }}
        {{ form.password }}
        {% if form.email.errors %}
          <ul class="errors">
            {% for error in form.password.errors %}
              <li>{{ error }}</li>{% endfor %}
          </ul>
        {% endif %}
      </fieldset>

      <div class="submit-button">
        {{ form.submit }}
      </div>

      <div class="login-signup">
        <span>Don't have an account? <a href="{{ url_for('auth_bp.signup') }}">Sign up.</a></span>
      </div>

    </form>
  </div>
{% endblock %}
```

The stage is set to start kicking some ass. Let's dig in.

Creating Our Login Routes

Let's get things started by setting up a Blueprint for our authentication-related routes:

auth.py

```
"""Routes for user authentication."""
from flask import Blueprint


# Blueprint Configuration
auth_bp = Blueprint(
    'auth_bp', __name__,
    template_folder='templates',
    static_folder='static'
)


@auth_bp.route('/login', methods=['GET', 'POST'])
def login():
    # Login route logic goes here


@auth_bp.route('/signup', methods=['GET', 'POST'])
def signup():
    # Signup logic goes here
```

Now we can start fleshing our signup and login routes. Before we can log anybody in, we need to make sure they can sign up.

Signing Up

The skeleton of our signup route needs to handle GET requests when users land on the page for the first time, and POST requests when users attempt to submit the signup form:

auth.py

```
...
from flask import Blueprint, render_template, request
from .forms import SignupForm

...

@auth_bp.route('/signup', methods=['GET', 'POST'])
def signup():
```

```
    """
    User sign-up page.

    GET requests serve sign-up page.
    POST requests validate form & user creation.
    """
    form = SignupForm()
    if form.validate_on_submit():
        # User sign-up logic will go here.
        ...
    return render_template(
        'signup.jinja2',
        title='Create an Account.',
        form=form,
        template='signup-page',
        body="Sign up for a user account."
    )
```

Our route will check all cases of a user attempting to sign in first by checking the HTTP method via Flask's **request** object. If the user is arriving for the first time, our route falls back to serve the **signup.jinja2** template via `render_template()`.

Time to introduce our first taste of **flask_login** logic:

auth.py

```
from flask import Blueprint, redirect, render_template, flash, request, session, url_for
from flask_login import login_required, logout_user, current_user, login_user
from .forms import LoginForm, SignupForm
from .models import db, User
from . import login_manager

...

@auth_bp.route('/signup', methods=['GET', 'POST'])
def signup():
    """
    User sign-up page.

    GET requests serve sign-up page.
    POST requests validate form & user creation.
    """
    form = SignupForm()
    if form.validate_on_submit():
        existing_user = User.query.filter_by(email=form.email.data).first()
        if existing_user is None:
            user = User(
                name=form.name.data,
                email=form.email.data,
                website=form.website.data
```

```
            )
            user.set_password(form.password.data)
            db.session.add(user)
            db.session.commit()  # Create new user
            login_user(user)  # Log in as newly created user
            return redirect(url_for('main_bp.dashboard'))
        flash('A user already exists with that email address.')
    return render_template(
        'signup.jinja2',
        title='Create an Account.',
        form=form,
        template='signup-page',
        body="Sign up for a user account."
    )
```

We first validate if the user filled out the form correctly via a useful built-in method: `form.validate_on_submit()` (`validate_on_submit()` will only trigger if the incoming request is a POST request containing form information, hence why we don't check that `request.method` is equal to `POST`). If any of our form's validators are not met, the user is redirected back to the signup form with error messages present.

We need to make sure the user isn't trying to sign up with an email which is taken by another user. Thanks to our User model, this is as simple as a single line:

Attempt to fetch a user with the provided email

```
existing_user = User.query.filter_by(email=form.email.data).first()
```

If our `existing_user` query yields no results (AKA: `is None`), we're clear to create a new user. Creating a user is as easy as passing some keyword arguments to our **User** model and generating a password via our model's `set_password()` method:

Create a user via our model.

```
...
user = User(
    name=form.name.data,
    email=form.email.data,
    website=form.website.data
)
user.set_password(form.password.data)
...
```

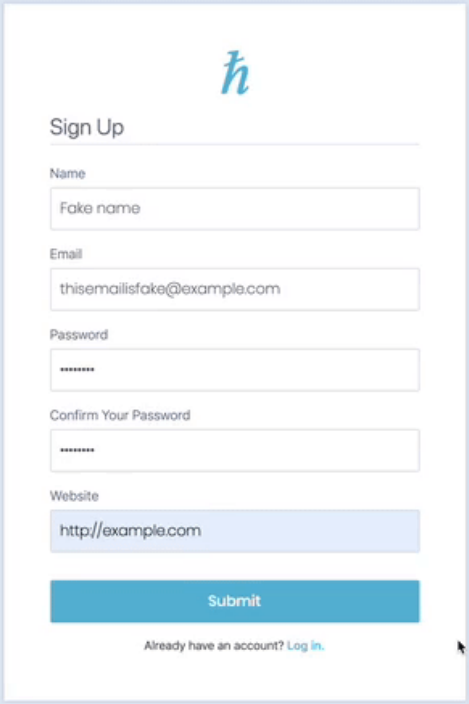
Our user is ready to be added to our database, at which point we can login them in:

Commit our new user record and log the user in.

```
...
db.session.add(user)
db.session.commit()  # Create new user
login_user(user)  # Log in as newly created user
...
```

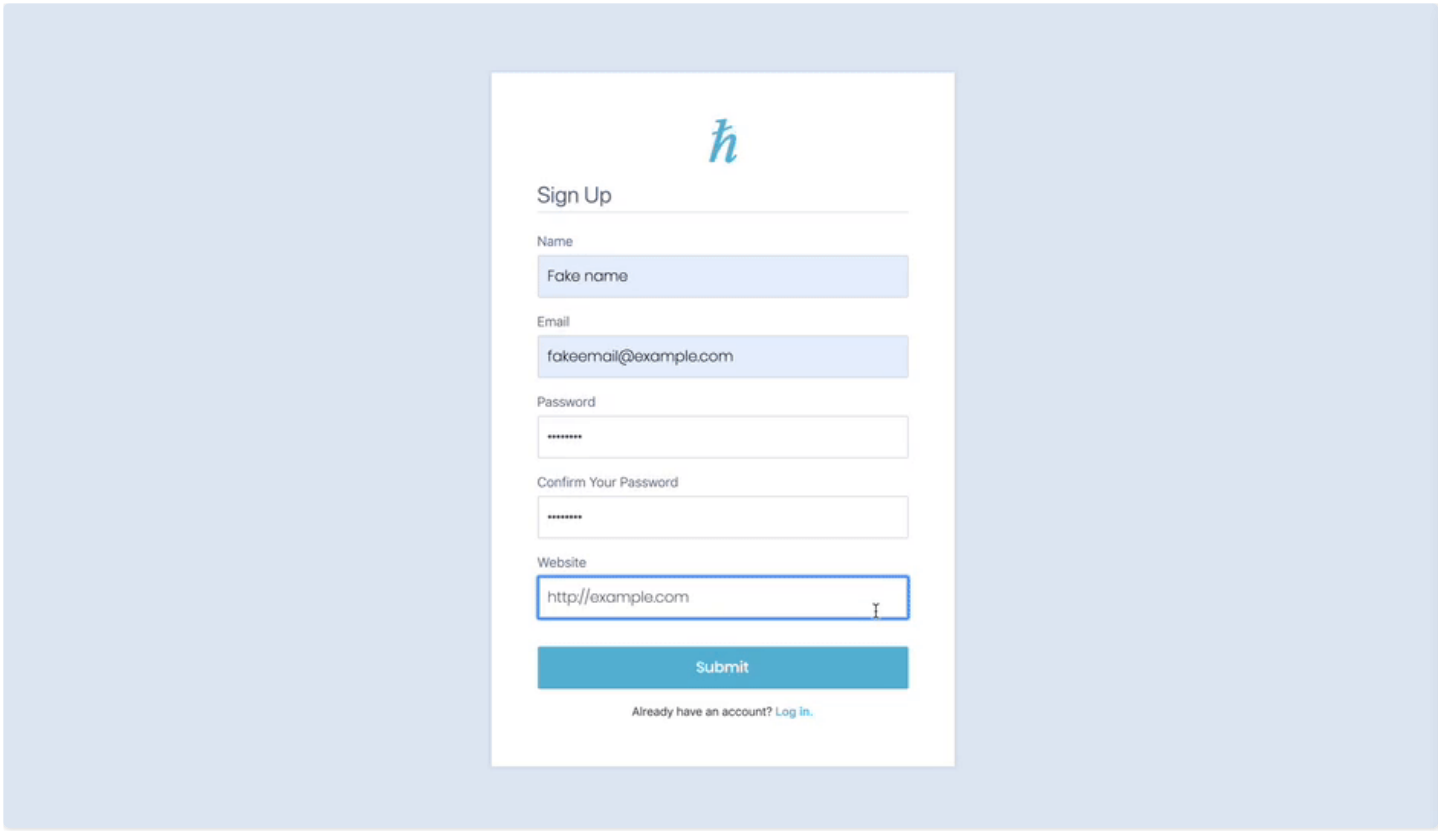
`login_user()` is a method that comes from the **flask_login** package that does exactly what it says: magically logs our user in. Flask-Login handles the nuances of everything this implies behind the scenes... all we need to know is that WE'RE IN!

If everything goes well, the user will finally be redirected to the main application. We handle this via `return redirect(url_for('main_bp.dashboard'))`:



A successful user login.

And here's what will happen if we log out and try to sign up with the same information:



Attempting to sign up with an existing email

Logging In

You'll be happy to hear that signup up was the hard part. Our login route shares much of the same logic we've already covered:

auth.py

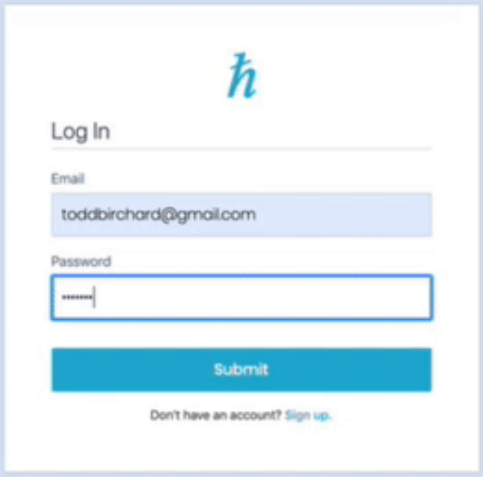
```
...

@auth_bp.route('/login', methods=['GET', 'POST'])
def login():
    """
    Log-in page for registered users.

    GET requests serve Log-in page.
    POST requests validate and redirect user to dashboard.
    """
    # Bypass if user is logged in
    if current_user.is_authenticated:
        return redirect(url_for('main_bp.dashboard'))

    form = LoginForm()
    # Validate login attempt
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user and user.check_password(password=form.password.data):
            login_user(user)
            next_page = request.args.get('next')
```

```
            return redirect(next_page or url_for('main_bp.dashboard'))
        flash('Invalid username/password combination')
        return redirect(url_for('auth_bp.login'))
    return render_template(
        'login.jinja2',
        form=form,
        title='Log in.',
        template='login-page',
        body="Log in with your User account."
    )
```

The login route is virtually identical to signing up until we check to see if the user exists. This time around a match results in *success* as opposed to a failure.

We then use our model's `user.check_password()` method to check the hashed password we created earlier.

As with last time, a successful login ends in `login_user(user)`. Our redirect logic is a little more sophisticated this time around: instead of always sending the user back to the dashboard, we check for `next`, which is a parameter stored in the query string of the current user (we're getting around to handling this in a bit). If the user attempted to access our app before logging in, `next` would equal the page they had tried to reach: this allows us wall-off our app from unauthorized users, and then drop users off at the page they tried to reach before they logged in:



A successful log in.

IMPORTANT: Login Helpers

Before your app can work like the above, we need to finish **auth.py** by providing a couple more routes:

auth.py

```
...

@login_manager.user_loader
def load_user(user_id):
    """Check if user is logged-in on every page load."""
    if user_id is not None:
        return User.query.get(user_id)
    return None


@login_manager.unauthorized_handler
def unauthorized():
    """Redirect unauthorized users to Login page."""
    flash('You must be logged in to view that page.')
    return redirect(url_for('auth_bp.login'))
```

`load_user` is critical for making our app work: before every page load, our app must verify whether or not the user is logged in (or *still* logged in after time has elapsed). `user_loader` loads users by their unique ID. If a user is returned, this signifies a logged-out user. Otherwise, when `None` is returned, the user is logged out.

Lastly, we have the `unauthorized` route, which uses the `unauthorized_handler` decorator for dealing with unauthorized users. Any time a user attempts to hit our app and is unauthorized, this route will fire.

Logged-in Routes

The last thing we'll cover is how to protect parts of our app from unauthorized users. Here's what we have in **routes.py**:

routes.py

```
"""Logged-in page routes."""
from flask import Blueprint, render_template, redirect, url_for
from flask_login import current_user, login_required


# Blueprint Configuration
main_bp = Blueprint(
```

```
    'main_bp', __name__,
    template_folder='templates',
    static_folder='static'
)


@main_bp.route('/', methods=['GET'])
@login_required
def dashboard():
    """Logged-in User Dashboard."""
    return render_template(
        'dashboard.jinja2',
        title='Flask-Login Tutorial.',
        template='dashboard-template',
        current_user=current_user,
        body="You are now logged in!"
    )
```

The magic here is all contained within the `@login_required` decorator. When this decorator is present on a route, the following things happen:

- The `@login_manager.user_loader` route we created determines whether or not the user is authorized to view the page (logged in). If the user is logged in, they'll be permitted to view the page.
- If the user is not logged in, the user will be redirected as per the logic in the route decorated with `@login_manager.unauthorized_handler`.
- The name of the route the user attempted to access will be stored in the URL as `?url=[name-of-route]` – this what allows `next` to work.

Logging Out

There's one last route to handle before we say goodbye. Coincidently, it's the route that figuratively allows our users to "say goodbye:"

routes.py

```
...
from flask_login import logout_user

...

@main_bp.route("/logout")
@login_required
def logout():
    """User log-out logic."""
    logout_user()
    return redirect(url_for('auth_bp.login'))
```

There You Have It

If you've made it this far, I commend you for your courage. To reward your accomplishments, I've published the source code for this tutorial on Github for your reference. Godspeed, brave adventurer.

toddbirchard/flasklogin-tutorial

:man_technologist: :key: Tutorial to accompany corresponding post on hackersandslackers.com - toddbirchard/flasklogin-tutorial

toddbirchard • GitHub

Flask | Software Development | Python

Todd Birchard

118 Posts | Website | Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

ExperimentalHypothesis
0 points
2 months ago

i love these tutorials. so well structured and showing best practices. amazing