TUTORIAL

How To Add Authentication to Your App with Flask-Login

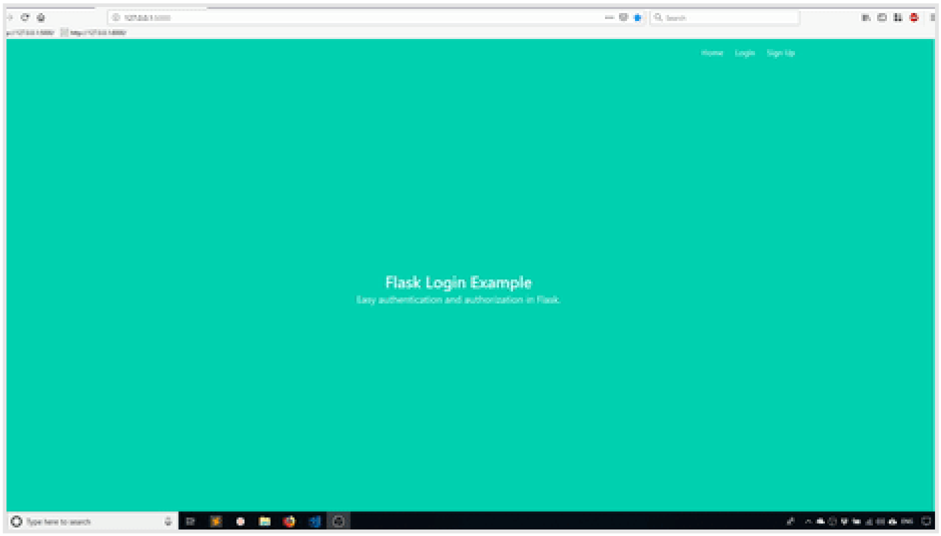
Python Python Frameworks

By Anthony Herbert

Updated July 6, 2020 84.1k

Introduction

Allowing users to log in to your app is one of the most common features you'll add to your web application. This article will cover how to add authentication to your Flask app with the Flask-Login package.



We're going to build some sign-up and login pages that allow users to log in and access protected pages that users who aren't logged in can't 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:

SCROLL TO TOP

- 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 log in
- Flash error messages back to users when something goes wrong
- Use information from the user's account to display on the profile page

The source code for this project is available on GitHub.

Prerequisites

To complete this tutorial, you will need the following:

- Python installed on a local environment.
- Knowledge of Basic Linux Navigation and File Management is helpful but not required.
- Familiarity with an editor like Visual Studio Code is helpful but not required.

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 for our regular routes, which include the index and the protected profile page. In a real app, you can break down the functionality in any way you like, but the solution covered here will work well for this tutorial.

Here is a diagram to provide a sense of what your project's file structure will look like once you have completed the tutorial:

```
.
└── flask_auth_app
    └── project
        ├── __init__.py       # setup our app
        ├── auth.py           # the auth routes for our app
        ├── db.sqlite         # our database
        ├── main.py           # the non-auth routes for our app
        ├── models.py         # our user model
        └── templates
```

SCROLL TO TOP

```
        ├── base.html     # contains common layout and links
        ├── index.html    # show the home page
        ├── login.html    # show the login form
        ├── profile.html  # show the profile page
        └── signup.html   # show the signup form
```

As we progress through the tutorial, we will create these directories and files.

Step 1 — Installing 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 be using SQLite to avoid having to install any extra dependencies for the database.

First, we will start with creating the project directory:

```
$ mkdir flask_auth_app
```

Next, we need to navigate to the project directory:

```
$ cd flask_auth_app
```

You will want to create a Python environment if you don't have one. Depending on how Python was installed on your machine, your commands will look similar to:

```
$ python3 -m venv auth
$ source auth/bin/activate
```

Note: You can consult the tutorial relevant to your local environment for setti SCROLL TO TOP

Run the following commands from your virtual environment to install the needed packages:

```
(auth)$ pip install flask flask-sqlalchemy flask-login
```

Now that you've installed the packages, you are ready to create the main app file.

Step 2 — Creating the Main App File

Let's start by creating a `project` directory:

```
(auth)$ mkdir project
```

The first file we will work on will be the `__init__.py` file for our project:

```
(auth)$ nano project/__init__.py
```

This file 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. We need to initialize SQLAlchemy, set some configuration values, and register our blueprints here.

project/__init__.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

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

def create_app():
    app = Flask(__name__)

    app.config['SECRET_KEY'] = 'secret-key-goes-here'
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite'

    db.init_app(app)

    # blueprint for auth routes in our app
    from .auth import auth as auth_blueprint
```

SCROLL TO TOP

```
    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
```

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

Step 3 — Adding 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 the login route.

For our auth blueprint, we'll have routes to retrieve both the login page (`/login`) and the sign-up page (`/sign-up`). We'll also have routes for handling the POST requests from both of those two routes. Finally, we'll have a logout route (`/logout`) to log out an active user.

For the time being, we will define `login`, `signup`, and `logout` with simple returns. We will revisit them at a later step and update them with desired functionality.

First, create `main.py` for your `main_blueprint`:

```
(auth)$ nano project/main.py
```

project/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'
```

SCROLL TO TOP

Next, create `auth.py` for your `auth_blueprint`:

```
(auth)$ nano project/auth.py
```

project/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'
```

In a terminal, you can set the `FLASK_APP` and `FLASK_DEBUG` values:

```
(auth)$ export FLASK_APP=project
(auth)$ export FLASK_DEBUG=1
```

The `FLASK_APP` environment variable instructs Flask how to load the app. It should point to where `create_app` is located. For our needs, we will be pointing to the `project` directory.

The `FLASK_DEBUG` environment variable is enabled by setting it to `1`. This will enable a debugger that will display application errors in the browser.

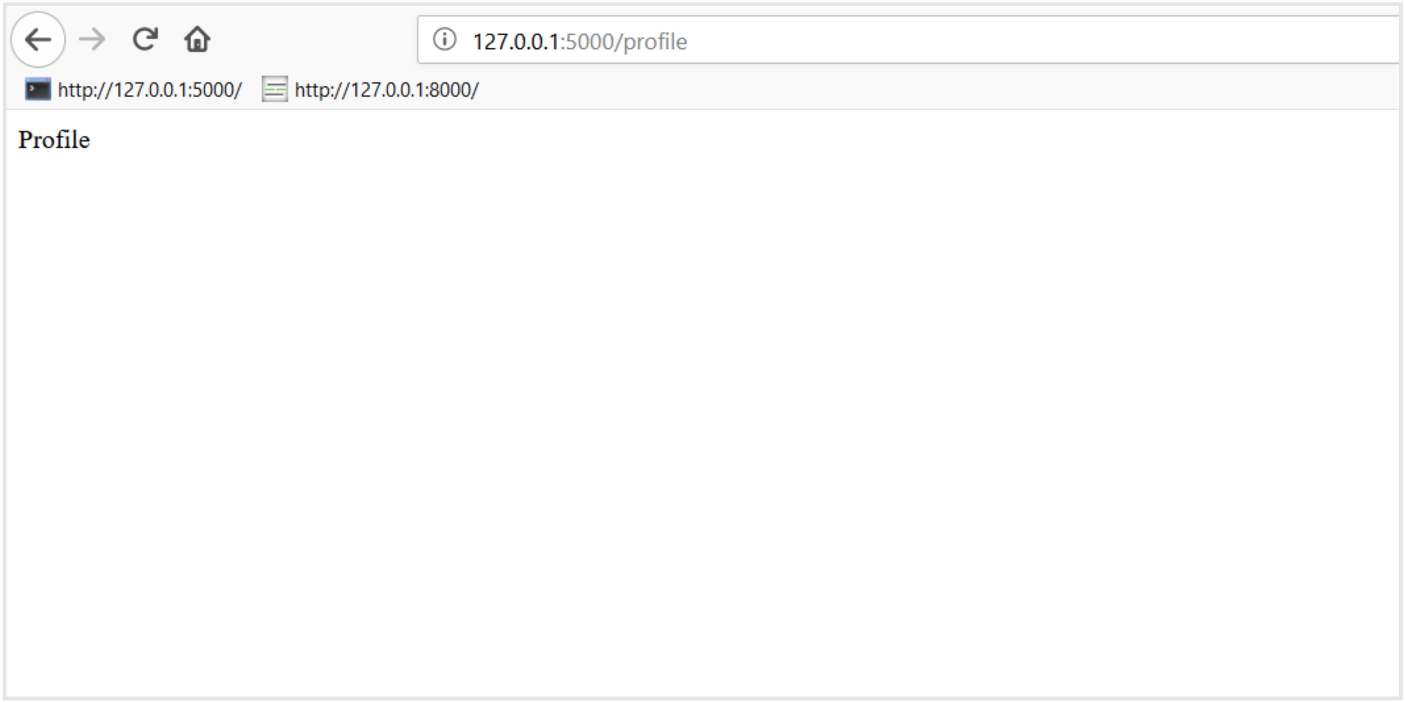
Ensure that you are in the `flask_auth_app` directory and then run the project:

```
(auth)$ flask run
```

SCROLL TO TOP

Now, in a web browser, you should be able to navigate to the five possible URLs and see the text returned that was defined in `auth.py` and `main.py`.

For example, visiting `localhost:5000/profile` displays: `Profile`:



Profile

Now that we have verified that our routes are behaving as expected, we can move on to creating templates.

Step 4 — Creating 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 SCROLL TO TOP create them now.

First, create a `templates` directory under the `project` directory:

```
(auth)$ mkdir -p project/templates
```

Then create `base.html`:

```
(auth)$ nano project/templates/base.html
```

Next, add the following code to the `base.html` file:

project/templates/base.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Flask Auth Example</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.2/css/bulm
</head>

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

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

                    <div id="navbarMenuHeroA" class="navbar-menu">
                        <div class="navbar-end">
                            <a href="{{ url_for('main.index') }}" class="navbar-item">
                                Home
                            </a>
                            <a href="{{ url_for('main.profile') }}" class="navbar-item">
                                Profile
                            </a>
                            <a href="{{ url_for('auth.login') }}" class="navbar-item">
                                Login
                            </a>
                            <a href="{{ url_for('auth.signup') }}" class="navbar-item">
                                Sign Up
                            </a>
                            <a href="{{ url_for('auth.logout') }}" class=
                                Logout
                            </a>
                        </div>
                    </div>
                </div>
            </nav>
        </div>

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

</html>
```

This code will create a series of menu links to each page of the application and an area where content will appear.

> **Note:** Behind the scenes, we are using Bulma to handle styling and layout. For a deeper dive into Bulma, consider reading the official Bulma documentation.

Next, create `templates/index.html`:

```
(auth)$ nano project/templates/index.html
```

Add the following code to the newly create file to add content to the page:

project/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 %}
```

SCROLL TO TOP

This code will create a basic index page with a title and subtitle.

Next, create `templates/login.html`:

```
(auth)$ nano project/templates/login.html
```

This code generates a login page with fields for **Email** and **Password**. There is also a checkbox to “remember” a logged in session.

project/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" type="email" name="email" placeholder="You
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large" type="password" name="password" placeholde
                </div>
            </div>
            <div class="field">
                <label class="checkbox">
                    <input type="checkbox">
                    Remember me
                </label>
            </div>
            <button class="button is-block is-info is-large is-fullwidth">Login</button>
        </form>
    </div>
</div>
{% endblock %}
```

Next, create `templates/signup.html`:

SCROLL TO TOP

```
(auth)$ nano project/templates/signup.html
```

Add the following code to create a sign-up page with fields for email, name, and password:

project/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" type="email" name="email" placeholder="Ema
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large" type="text" name="name" placeholder="Name'
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large" type="password" name="password" placeholde
                </div>
            </div>

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

Next, create `templates/profile.html`:

```
(auth)$ nano project/templates/profile.html
```

Add this code to create a simple page with a title that is hardcoded to welcome **Anthony**:

SCROLL TO TOP

project/templates/profile.html

```
{% extends "base.html" %}

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

Later, we will add code to dynamically greet any user.

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.

Next, update `main.py` by modifying the import line and the routes for `index` and `profile`:

project/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')
```

Now you will update `auth.py` by modifying the import line and routes for `login` and `signup`:

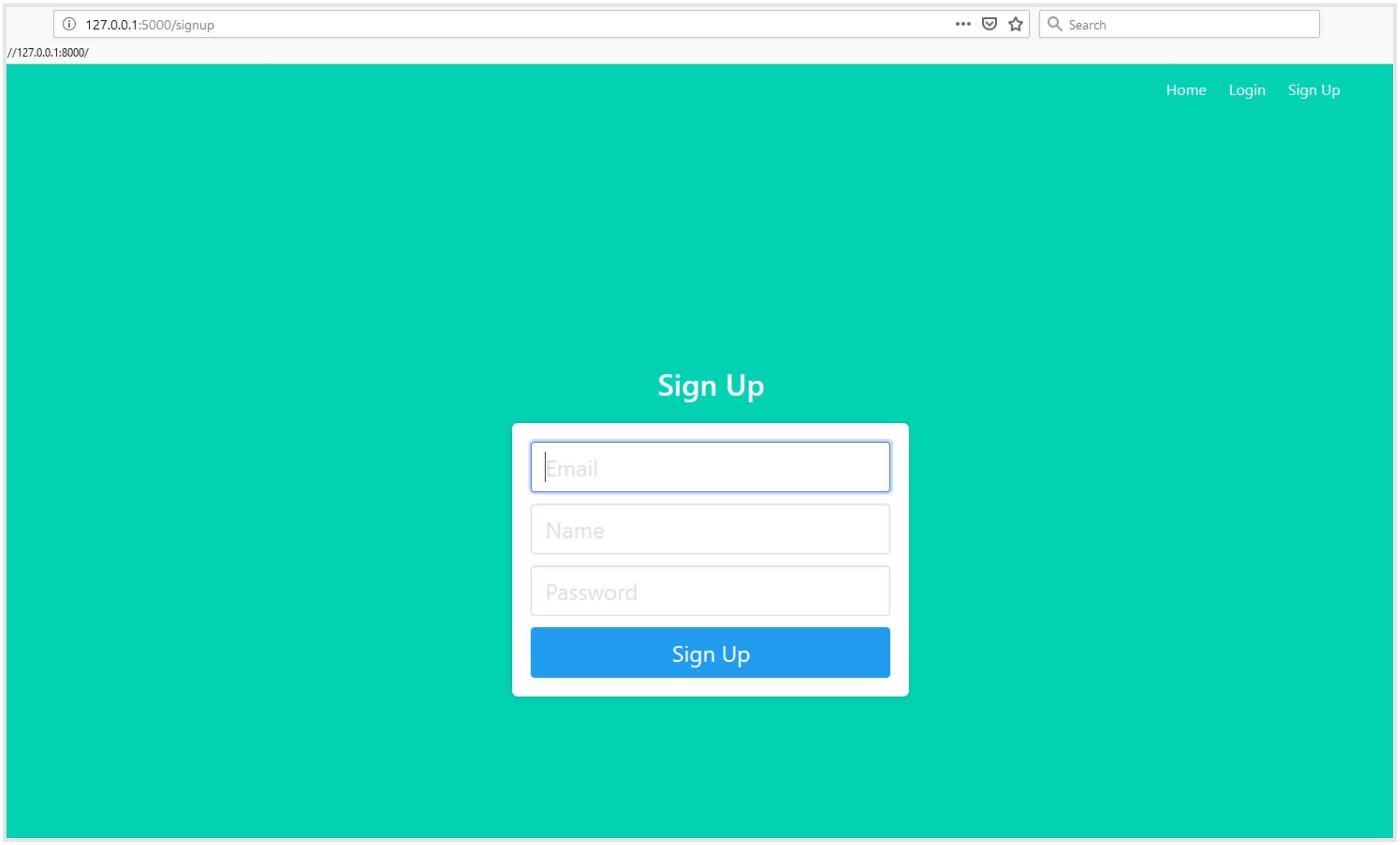
project/auth.py

```
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')
```

SCROLL TO TOP

Once you've made these changes, here is what the sign-up page looks like if you navigate to `/sign-up`:



You should be able to 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.

Step 5 — Creating User Models

Our user model represents what it means for our app to have a user. We'll have fields for an email address, password, and name. 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 that 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:

SCROLL TO TOP

```
(auth)$ nano project/models.py
```

This code creates a user model with columns for an `id`, `email`, `password`, and `name`:

project/models.py

```
from . import db

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

Now that you've created a user model, you can move on to configuring your database.

Step 6 — Configuring the Database

As stated in the Prerequisites, 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 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. Ensure that you are still in the virtual environment and in the `flask_auth_app` directory.

```
>>> from project import db, create_app
>>> db.create_all(app=create_app()) # pass the create_app result so Flask-SQLAlchemy gets the
```

Note: If using the Python interpreter is new to you, you can consult the official documentation.

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

SCROLL TO TOP

Step 7 — Setting Up the Authorization Function

For our sign up function, we're going to take the data the user types into the form and add it to our database. 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 POST form data. In this function, we will gather the data passed from the user first.

Create the function and add a redirect to the bottom. This will provide a user experience of a successful signup and being directed to the Login Page.

Update `auth.py` by modifying the import line and implementing `signup_post`:

project/auth.py

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

Continue to update `auth.py` by adding imports and implementing `signup_post`:

auth.py

```
from flask import Blueprint, render_template, redirect, url_for, request
from werkzeug.security import generate_password_hash, check_password_hash
from .models import User
from . import db
...
@auth.route('/signup', methods=['POST'])
def signup_post():
    email = request.form.get('email')
```

SCROLL TO TOP

```
    name = request.form.get('name')
    password = request.form.get('password')

    user = User.query.filter_by(email=email).first() # if this returns a user, then the emai

    if user: # if a user is found, we want to redirect back to signup page so user can try a
        return redirect(url_for('auth.signup'))

    # create a new user with the form data. Hash the password so the plaintext version isn't
    new_user = User(email=email, name=name, password=generate_password_hash(password, methoc

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

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

Note: Storing passwords in plaintext is considered a poor security practice. You will generally want to utilize a complex hashing algorithm and a password salt to keep passwords secure.

Step 8 — Testing the 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 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 sign-up page.

project/auth.py

SCROLL TO TOP

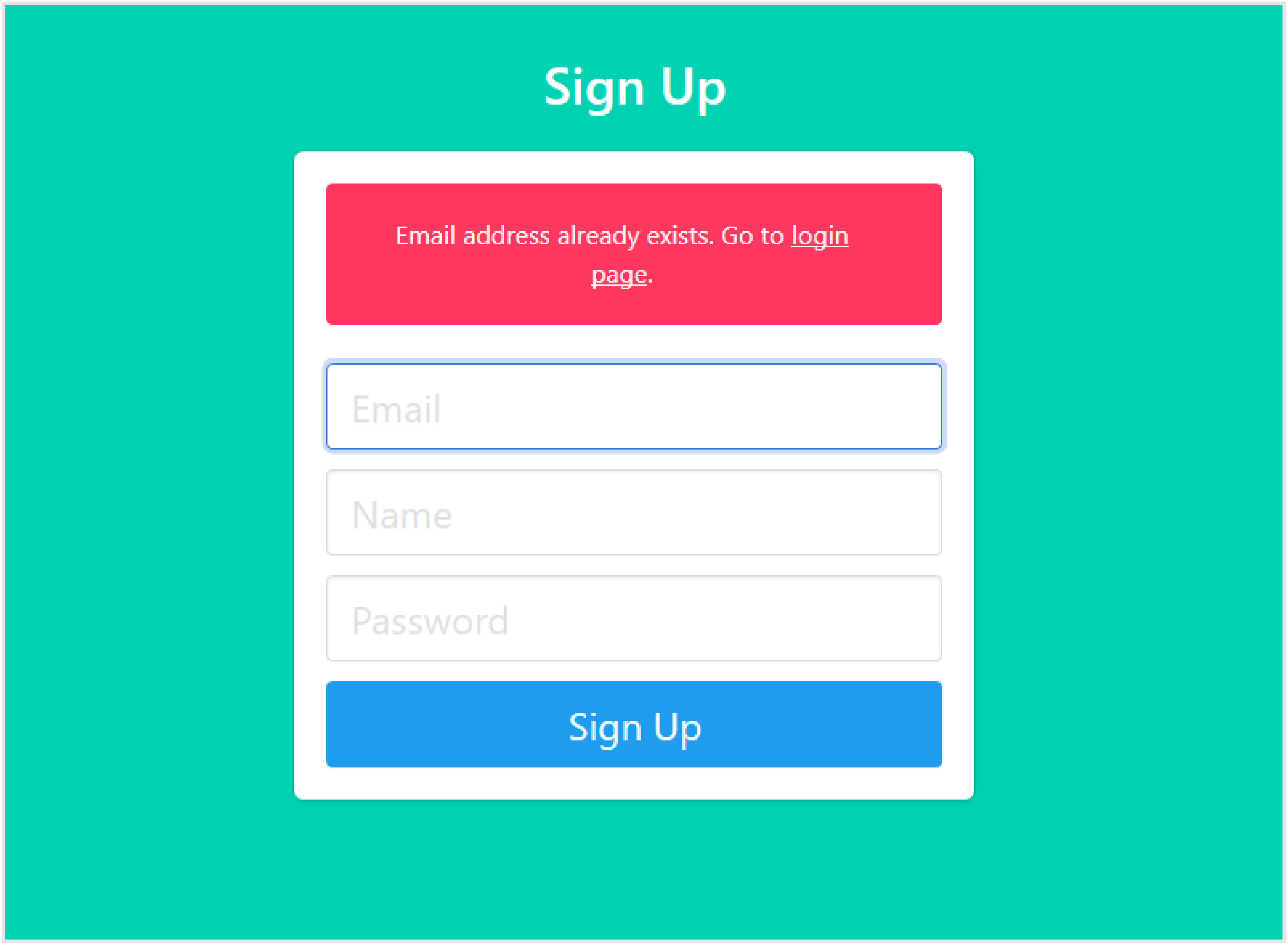
```
from flask import Blueprint, render_template, redirect, url_for, request, flash
...
@auth.route('/signup', methods=['POST'])
def signup_post():
    ...
    if user: # if a user is found, we want to redirect back to signup page so user can try a
        flash('Email address already exists')
        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.

project/templates/signup.html

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

SCROLL TO TOP



Step 9 — Adding the Login Method

The login method is similar to the sign-up 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 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:

SCROLL TO TOP

project/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:

project/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') else False

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

    # check if the user actually exists
    # take the user-supplied password, hash it, and compare it to the hashed password in the
    if not user or not check_password_hash(user.password, password):
        flash('Please check your login details and try again.')
        return redirect(url_for('auth.login')) # if the user doesn't exist or password is wr

    # if the above check passes, then we know the user has the right credentials
    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 sign-up form, let's add the potential error message directly above the form:

project/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">
```

SCROLL TO TOP

We now have the ability to say a user has been logged in successfully, but there is nothing to log the user into. This is where we bring in Flask-Login to manage user sessions.

Before we get started, we need a few things for Flask-Login to work. Start by adding the `UserMixin` to your User model. The `UserMixin` will add Flask-Login attributes to the model so that Flask-Login will be able to work with it.

models.py

```
from flask_login import UserMixin
from . import db

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True) # primary keys are required by SQLAlchemy
    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 `init` code for Flask-Login:

project/__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 key of our user table, use it in the query f
        return User.query.get(int(user_id))
```

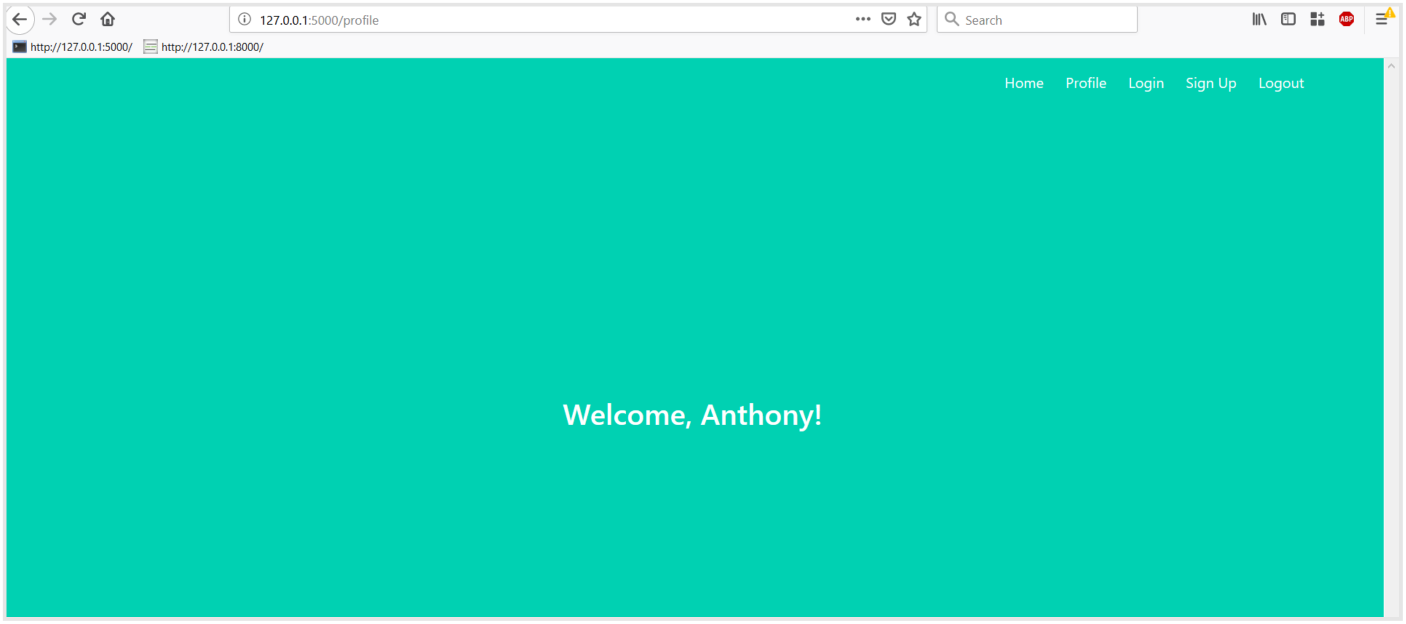
SCROLL TO TOP

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

project/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 user has the right credentials
    login_user(user, remember=remember)
    return redirect(url_for('main.profile'))
```

With Flask-Login setup, we can use the `/login` route. When everything is in place, you will see the profile page.



Welcome, Anthony!

Step 10 — Protecting 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, we add the `@login_requried` d SCROLL TO TOP 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 will then use that name and display its value.

project/main.py

```
from flask_login import login_required, current_user
...
@main.route('/profile')
@login_required
def profile():
    return render_template('profile.html', name=current_user.name)
```

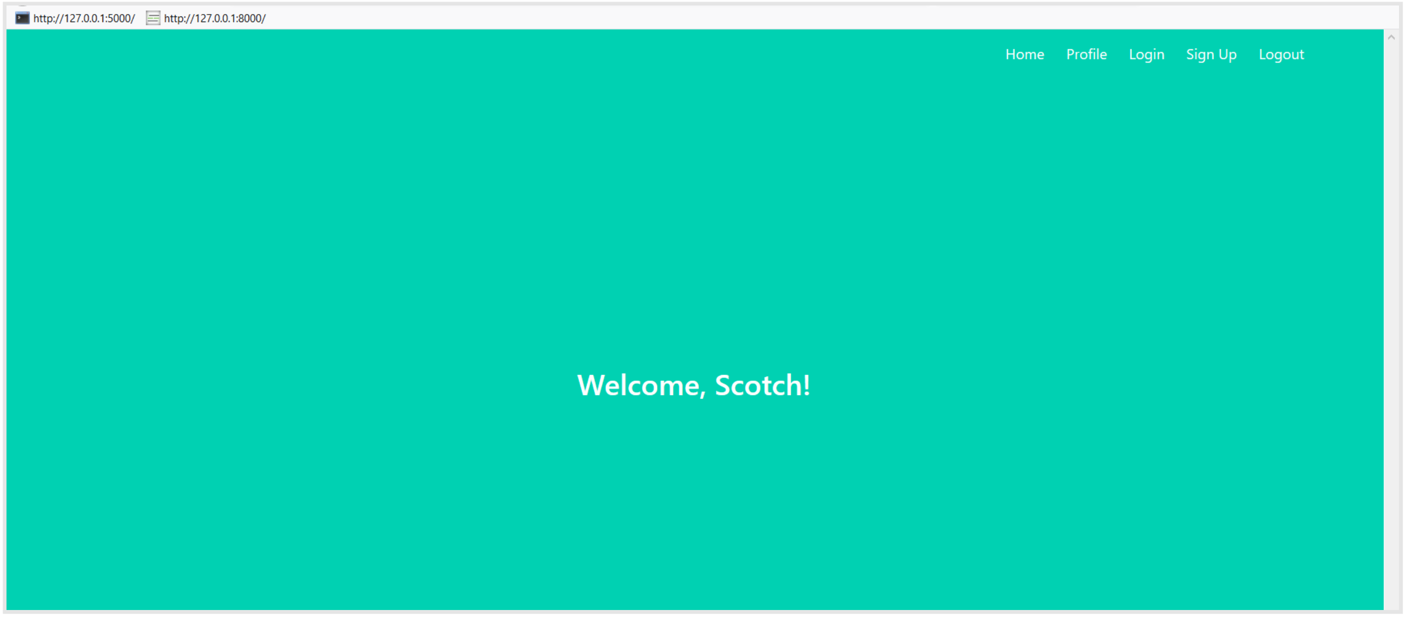
Then in the `profile.html` file, update the page to display the `name` value:

project/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.

SCROLL TO TOP



Welcome, Scotch!

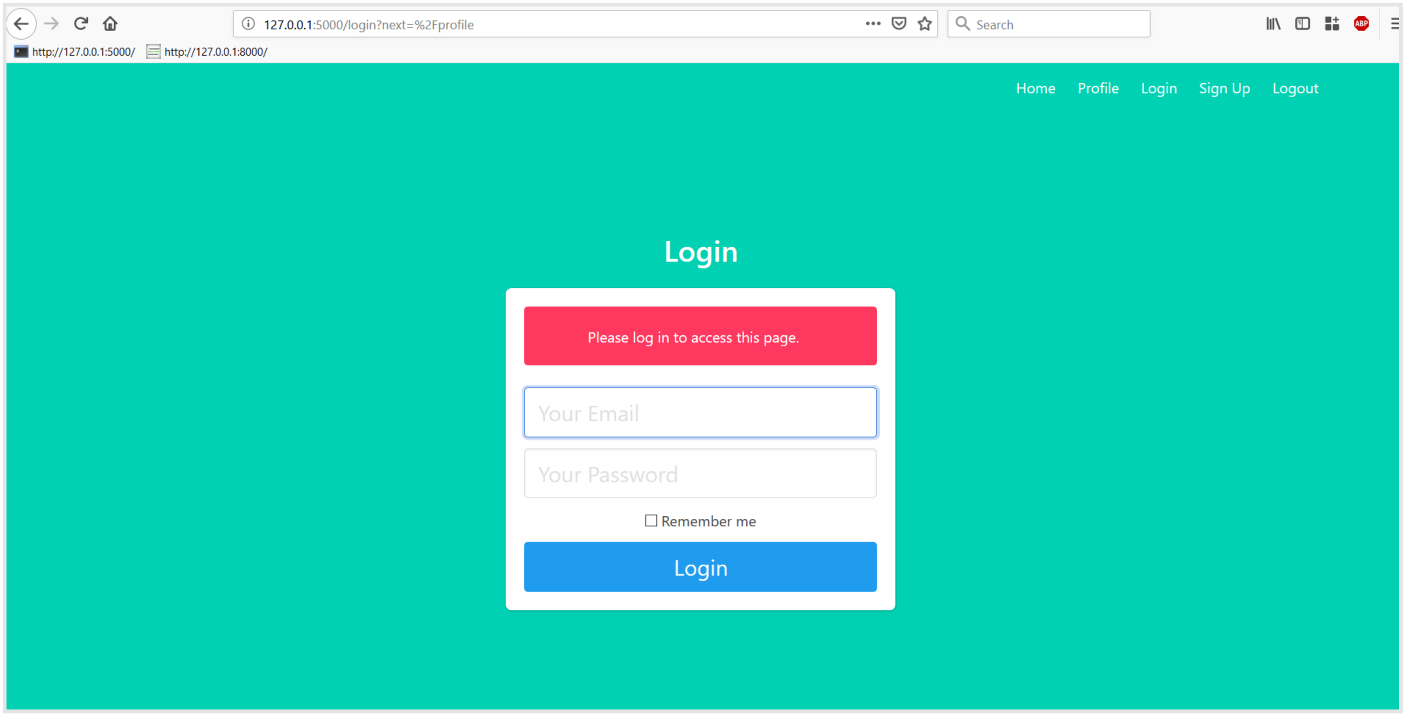
The final thing we can do is update the logout view. We can call the `logout_user` function in a route for logging out. We have the `@login_required` decorator because it doesn't make sense to logout a user who isn't logged in to begin with.

project/auth.py

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

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

SCROLL TO TOP



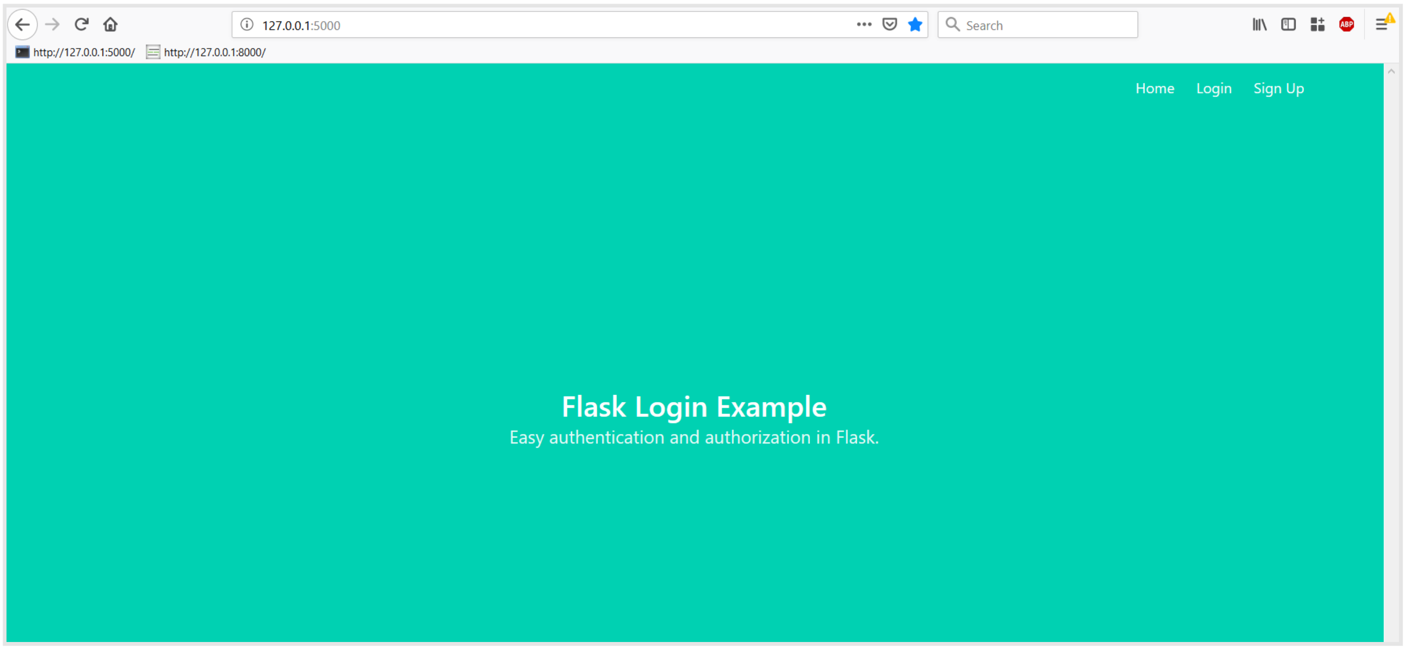
One last thing we can do is put `if` statements in the templates to display only the links relevant to the user. So before the user logs in, they will have the option to log in or sign up. After they have logged in, they can go to their profile or log out:

templates/base.html

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

SCROLL TO TOP

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



With that, you have successfully built your app with authentication.

Conclusion

We've used Flask-Login and Flask-SQLAlchemy to build a login system for our app. We covered how to authenticate a user by first creating a user model and storing the user information. 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` decorator 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 built on top of the Flask-Login library.

Was this helpful? Yes No 14

SCROLL TO TOP