Whale Ambience unlocked!

CONTENTS

// TUTORIAL //

Whale Ambience unlocked!

How To Add Authentication to Your App with Flask-Login

Updated on November 22, 2021

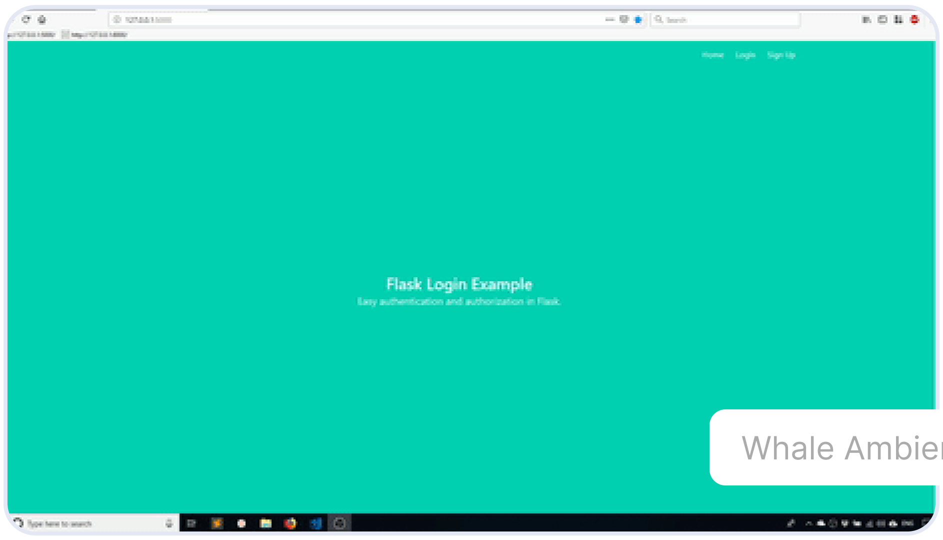
Python Python Frameworks

Anthony Herbert

English

Introduction

Allowing users to log in to your app is one of the most common features you will add to your web applications. You can add authentication to your Flask app with the Flask-Login package.



Whale Ambience unlocked!

In this tutorial, you will:

- Use the Flask-Login library for session management
- Use the built-in Flask utility for hashing passwords
- Add protected pages to the app for logged in users only
- Use Flask-SQLAlchemy to create a `User` model
- Create sign-up and login forms for the 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

You will build a sign-up and a login page that allow users to log in and access protected pages. You will use information from the `User` model and display it on the protected pages when the user logs in to simulate what a profile would look like.

Note: This tutorial is limited in scope and does not cover advanced persisting of sessions. Furthermore, modifying the data type for the primary key or considerations for migrating to different database systems are also outside of the scope of this introductory tutorial.

The source code for this project is available on GitHub.

Prerequisites

To complete this tutorial, you will need the following:

- e familiarity with Python.
- P n installed on a local environment.
- Knowledge of Basic Linux Navigation and File Management.

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

```
.
└── flask_auth_app
    └── project
        ├── __init__.py       # setup the app
        ├── auth.py           # the auth routes for the app
        ├── db.sqlite         # the database
        ├── main.py           # the non-auth routes for the app
        ├── models.py         # the user model
        └── templates
            ├── base.html     # contains common layou
            ├── index.html    # show the home page
            ├── login.html    # show the login form
            ├── profile.html  # show the profile page
            └── signup.html   # show the signup form
```

Whale Ambience unlocked!

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

This tutorial was verified with `sqlite3` v3.36.0, `python` v3.9.8, `flask` v2.0.2, `flask-login` v0.5.0, and `flask-sqlachemy` v2.5.1.

Step 1 – Installing Packages

There are three main packages you need for your project:

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

You will be using SQLite to avoid having to install any extra dependencies for the database.

First, start with creating the project directory:

```
$ mkdir flask_auth_app
```

Copy

Next, navigate to the project directory:

```
$ c     lask_auth_app
```

Copy

You will want to create a Python environment if you don't have one.

> **Note:** You can consult the tutorial relevant to your local environment for setting up `venv`.

Depending on how Python was installed on your machine, your command will look similar to:

```
$ python3 -m venv  auth
```
Copy

Whale Ambience unlocked!

The `-m` flag is for `module-name`. This command will execute th
new virtual environment named `auth`. This will create a new directory containing `bin`, `include`, and `lib` subdirectories. And a `pyvenv.cfg` file.

Next, run the following command:

```
$ source  auth /bin/activate
```
Copy

This command will activate the virtual environment.

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

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

Now that you have 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
```
Copy

The first file will be the `__init__.py` file for the project:

```
(a[illegible] nano project/__init__.py
```
Copy

This app will use the Flask app factory pattern with blueprints. One blueprint handles the regular routes, which include the index and the protected profile page. Another blueprint handles everything auth-related. 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.

This file will have the function to create the app, which will initialize the database and register the blueprints. At the moment, this will not do much, but it will be needed for the rest of the app.

You will need to initialize SQLAlchemy, set some configuration values, and register the blueprints here:

Whale Ambience unlocked!

project/__init__.py

Copy

```
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():
```

Products

Solutions

Developers

Partners

Pricing

DigitalOcean

Log in

Sign up

Blog

Docs

Get Support

Then add your `main_blueprint`:

project/main.py

Whale Ambience unlocked!

```
from flask import Blueprint
from . import db

main = Blueprint('main', __name__)

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

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

For the `auth_blueprint`, you will have routes to retrieve both the login page (`/login`) and the sign-up page (`/signup`). Finally, you will have a logout route (`/logout`) to log out an active user.

Next, create `auth.py`:

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

Copy

Then add your `auth_blueprint`:

project/auth.py

Copy

```
from flask import Blueprint
from . import db

au[illegible]lueprint('auth', __name__)

@auth.route('/login')
```

```
def login():
    return 'Login'

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

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

For the time being, define `login`, `signup`, and `logout` with te
routes for handling the POST requests from `login` and `signu`
later and update it with the desired functionality.

Whale Ambience unlocked!

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

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

Copy

The `FLASK_APP` environment variable instructs Flask on how to load the app. You would want this to point to where `create_app` is located. For this tutorial, you 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.

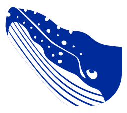
Ensure that you are in the `flask_auth_app` directory and then run the project:

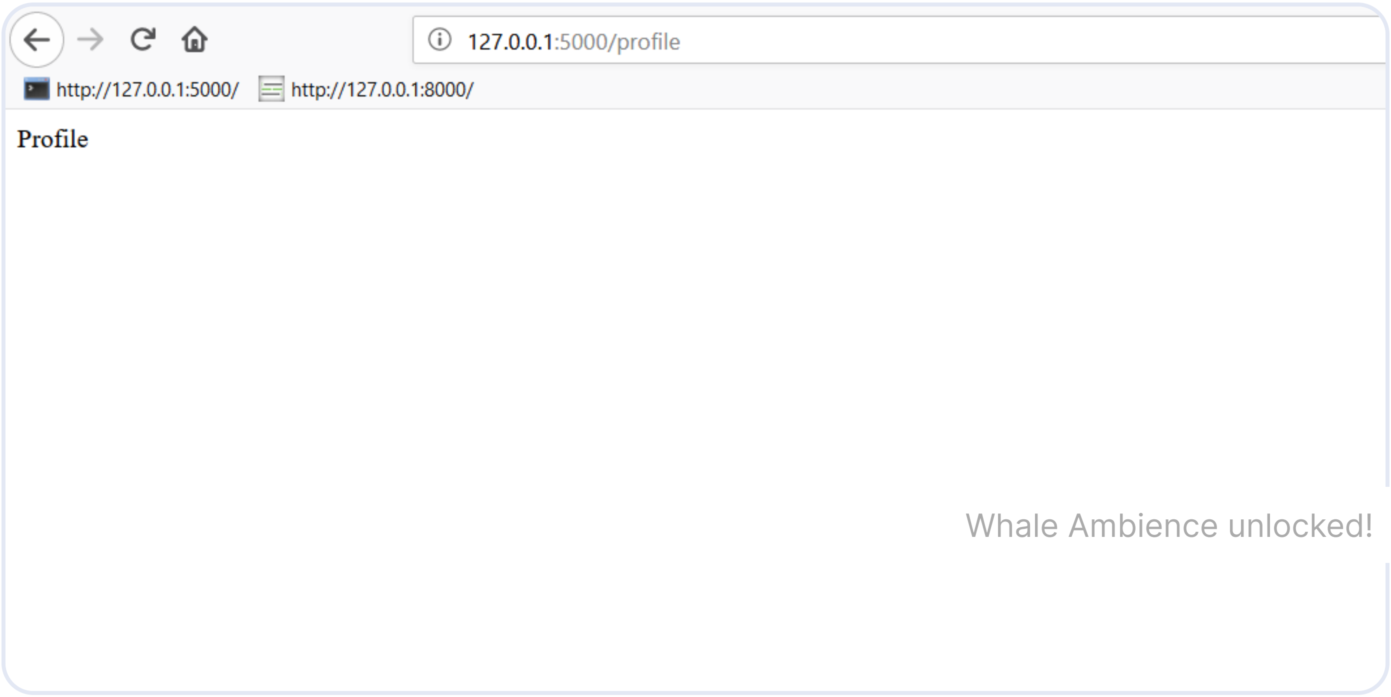
```
(auth)$ flask run
```

Copy

Now, in a web browser, you can 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

Whale Ambience unlocked!

Once you have verified that the routes are behaving as expected, you can create the templates.

Step 4 – Creating Templates

Next, create the templates that are used in the app. This is the first step before you can implement the actual login functionality.

The app will use four templates:

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

You will 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.

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

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

Then create `base.html`:

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

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=ed
    <meta name="viewport" content="width=device-width
    <title>Flask Auth Example</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.
</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-
                                Home
                            </a>
                            <a href="{{ url_for('main.profile') }}" class="navba
                                Profile
                            </a>
                            <a href="{{ url_for('auth.login') }}" class="navbar-
                                Login
                            </a>
                            <a href="{{ url_for('auth.signup') }}" class="navbar
                                Sign Up
                            </a>
                            <a href="{{ url_for('auth.logout') }}" class="navbar
                                Logout
                            </a>
                        </div>
                    </div>
                </div>
            </nav>
        </div>

        <div class="hero-body">
            <div class="container has-text-centered">
```

Whale Ambience unlocked!

```
                {% block content %}
                {% endblock %}
            </div>
        </div>
    </section>
</body>

</html>
```

This code will create a series of menu links to each page of the application. It also establishes a block for `content` that can be overwritten by chi

Whale Ambience unlocked!

> **Note:** This tutorial uses 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
```

Copy

Add the following code to the newly created 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 %}
```

Copy

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

Next, create `templates/login.html`:

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

Copy

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" plac
                </div>
            </div>

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

Copy

Whale Ambience unlocked!

Next, create `templates/signup.html`:

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

Copy

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

project/templates/signup.html

Copy

```
{% 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" plac
                </div>
            </div>

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

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

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

Whale Ambience unlocked!

Next, create `templates/profile.html`:

Copy

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

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

project/templates/profile.html

Copy

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

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

```
  Welcome, Anthony!
</h1>
{% endblock %}
```

You will revisit this code later to dynamically greet any user.

Once you have added the templates, you can update the return statements in each of the routes 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

Whale Ambience unlocked!

Copy

```
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`:

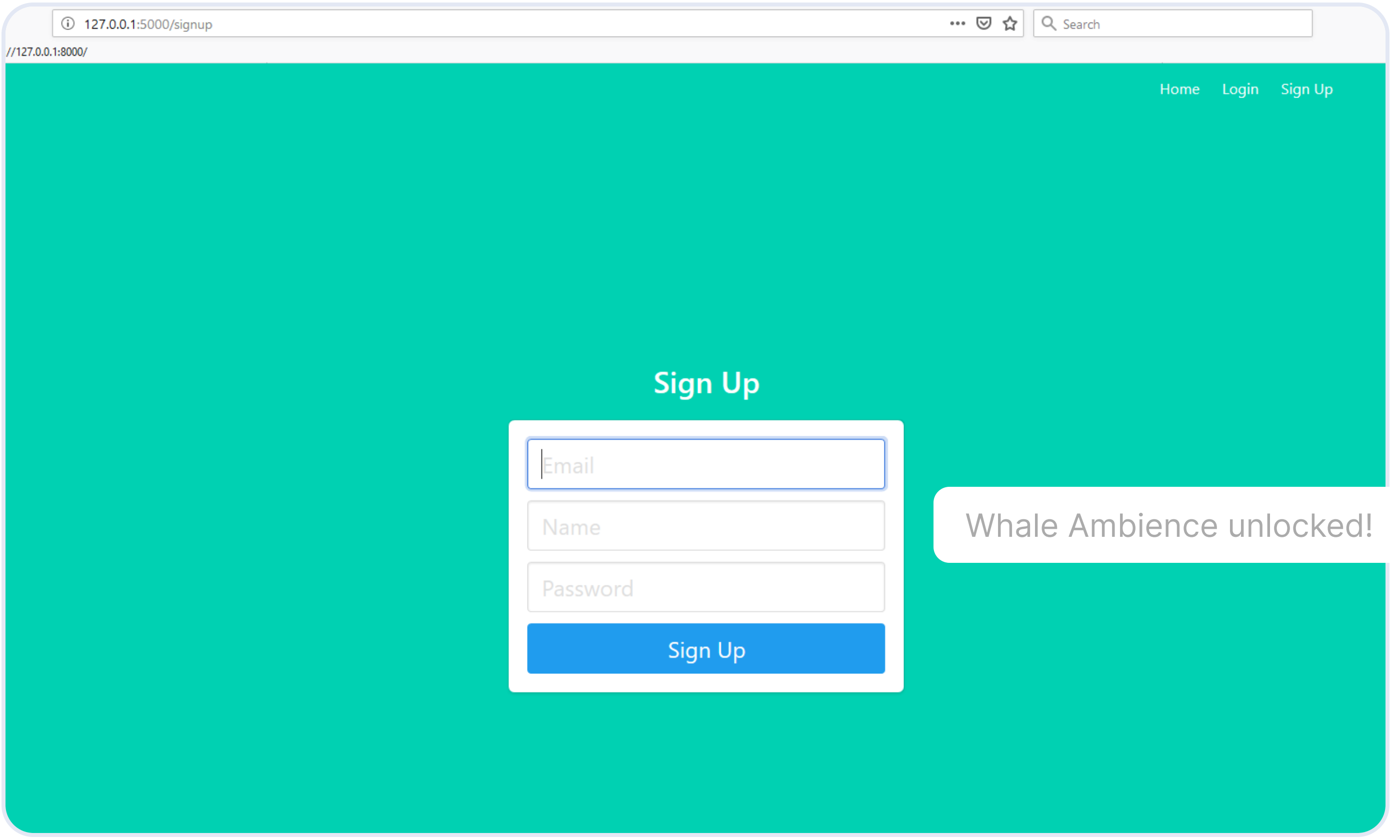
project/auth.py

Copy

```
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')
```

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



Whale Ambience unlocked!

You can navigate to the pages for `/`, `/login`, and `/profile` as well.

Leave `/logout` alone for now because it will not display a template later.

Step 5 – Creating User Models

The user model represents what it means for the app to have a user. This tutorial will require fields for an `email` address, `password`, and `name`. In future applications, you may decide you want much more information to be stored per user. You can add things like birthdays, profile pictures, locations, 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.

Create the `User` model:

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

Copy

Define the `User` model:

project/models.py

```
from . import db

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

Copy

This code defines a `User` with columns for an `id`, `email`, `pas` Whale Ambience unlocked!

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

Step 6 – Configuring the Database

You will be using an SQLite database. You could create an SQLite database on your own, but let's have Flask-SQLAlchemy do it for you. You already have the path of the database specified in the `__init__.py` file, so you will need to tell Flask-SQLAlchemy to create the database in the Python REPL.

Ensure that you are still in the virtual environment and in the `flask_auth_app` directory.

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

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

Copy

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 the user table in it.

Ste[illegible] – Setting Up the Authorization Function

For the sign-up function, you will take the data the user submits to the form and add it to the database. You will need to make sure a user with the same email address does not already exist in the database. If it does not exist, then you need to make sure you hash the password before placing it into the database.

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

Let's start by adding a second function to handle the POST form data. Gather the data passed from the user.

Whale Ambience unlocked!

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')
def signup():
    return render_template('signup.html')

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

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

Now, let's add the rest of the code necessary for signing up a user. 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 , reques
 from werkzeug.security import generate_password_hash, check_password_hash
[illegible]models import User
fro[illegible]mport db
...
@auth.route('/signup', methods=['POST'])
```

```
def signup_post():
    # code to validate and add user to database goes here
    email = request.form.get('email')
    name = request.form.get('name')
    password = request.form.get('password')

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

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

    # create a new user with the form data. Hash the password so the plaintext v
    new_user = User(email=email, name=name, password

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

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

Whale Ambience unlocked!

This code will check to see if a user with the same email address exists in the database.

Step 8 – Testing the Sign Up Method

Now that you have the sign-up method completed, you will be able to create a new user

Let’s test the form to create a user.

There are two ways you can verify if the sign-up was successful:

- 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.

Let’s add code to let the user know the email already exists and direct them to go to the login page. By calling the `flash` function, you can send a message to the next request, which in this case, is the redirect. The page the user is redirected to will then have access to that message in the template.

First, add the `flash` before you redirect to the sign-up page.

project/auth.py

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

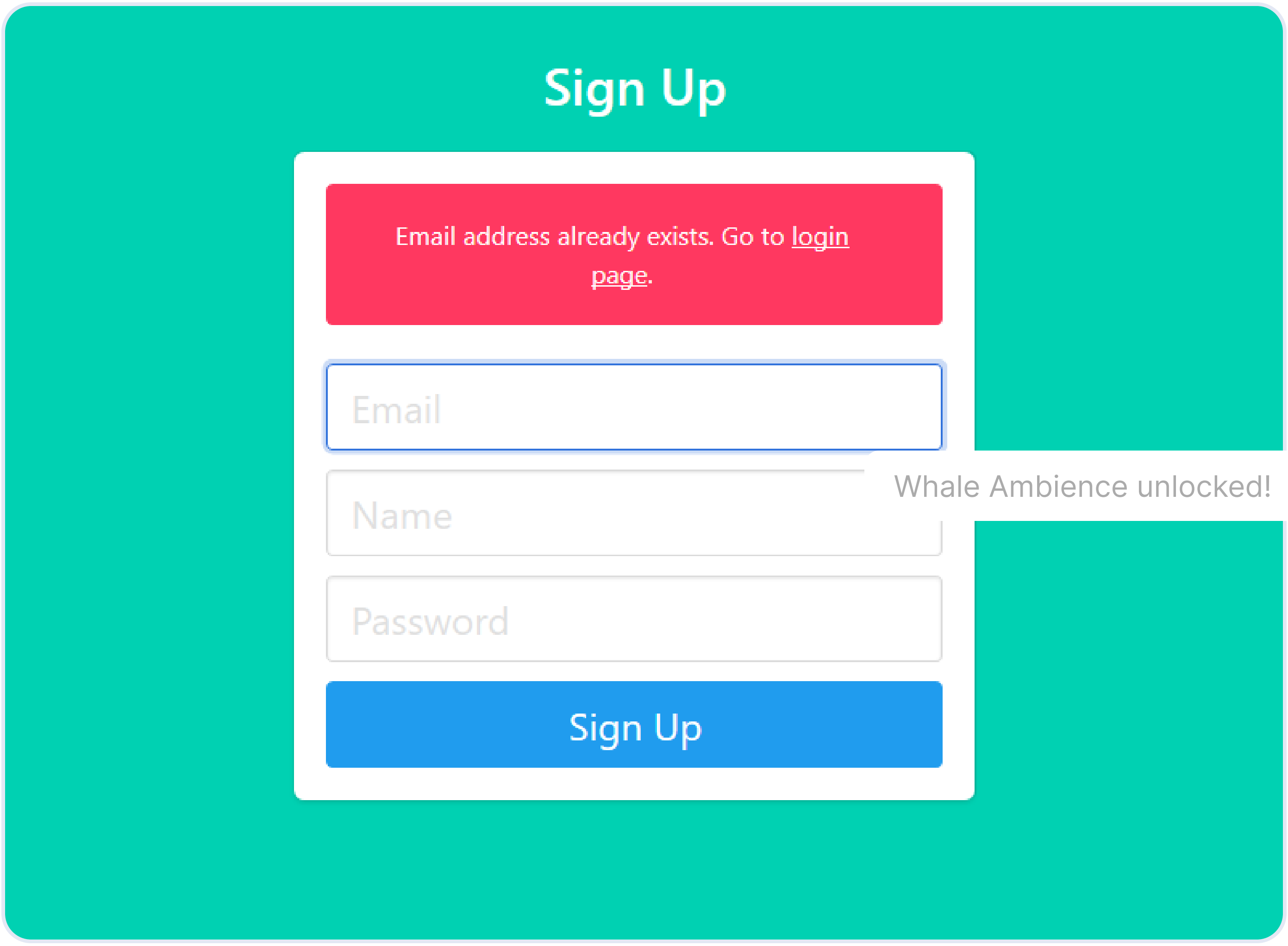
To get the flashed message in the template, you can add this code before the form

Whale Ambience unlocked!

project/templates/signup.html

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

This code will display the message `"Email address already exists. Go to login page."` if the email address is already in the database.



Email address already exists. Go to login page.

Whale Ambience unlocked!

At this point, you can run the application and attempt to sign up with an email address that already exists.

Step 9 – Adding the Login Method

The login method is similar to the sign-up function. In this case, you will compare the `email` address entered to see if it is in the database. If so, you 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. You will know the user has entered the correct `password` when both hashed `password`s match.

Once the user has passed the password check, you will know that they have the correct credentials and you 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.

You c[illegible]rt with a new route for handling the data submitted with POST. And redirect to the pro[illegible] page when the user successfully logs in:

project/auth.py

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

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

Copy

Whale Ambience unlocked!

Now, you need to verify if the user has the correct credentials:

project/auth.py

```
...
@auth.route('/login', methods=['POST'])
def login_post():
    # login code goes here
     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 pas
     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

    # if the above check passes, then we know the user has the right credentials
    return redirect(url_for('main.profile'))
```

Copy

Let’s add in the block in the template so the user can see the flashed message:

project/templates/login.html

```
...
 {% with messages = get_flashed_messages() %}
 {% if messages %}
```

Copy

```
    <div class="notification is-danger">
        {{ messages[0] }}
    </div>
{% endif %}
{% endwith %}
<form method="POST" action="/login">
```

You now have the ability to say a user has been logged in successfully, but there is nothing to log the user into.

Flask-Login can manage user sessions. Start by adding the `UserMixin` to your User model. The `UserMixin` will add Flask-Login attributes to the model sc Whale Ambience unlocked! 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
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(1000))
```

Then, you need to specify the *user loader*. A user loader tells Flask-Login how to find a specific user from the ID that is stored in their session cookie. Add this in the `create_app` function along with `init` code for Flask-Login:

project/__init__.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager
...
def create_app():
    ...
    db.init_app(app)

      gin_manager = LoginManager()
      in_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
         return User.query.get(int(user_id))
```

Finally, add the `login_user` function before redirecting to the profile page to create the session:

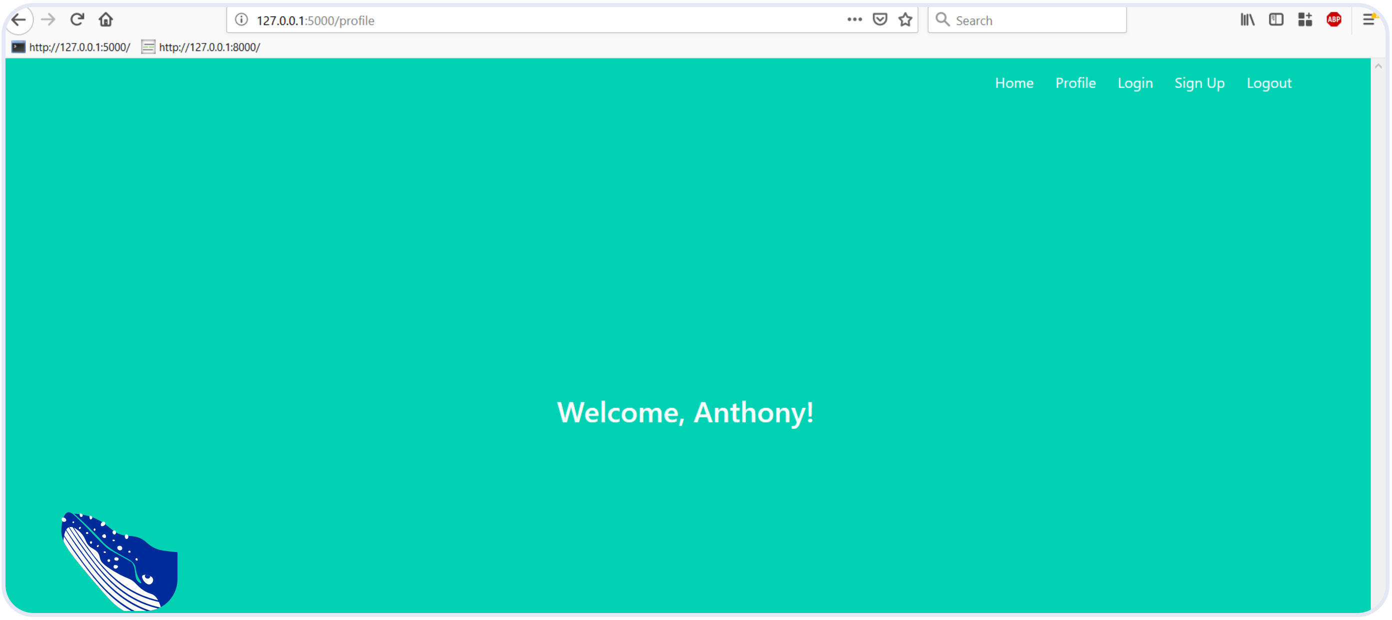
Whale Ambience unlocked!

project/auth.py

```
 from flask_login import login_user
from .models import User
from . import db
...
@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'))
```

Copy

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



Welcome, Anthony!

At this point, you can run the application and attempt to log in.

Step 10 – Protecting Pages

If your name is not **Anthony**, then you will see that your name is wrong on the profile page. The goal is for the profile to display the name in the database. You will need to protect the page and then access the user's data to get the `name`.

To protect a page when using Flask-Login, add the `@login_requried` decorator between the route and the function. This will prevent a user that is not logged in from seeing the route. If the user is not logged in, the user will get redirected t Whale Ambience unlocked! Flask-Login configuration.

With routes that are decorated with the `@login_required` decorator, you can use the `current_user` object inside of the function. This `current_user` represents the user from the database and provides 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:

project/main.py

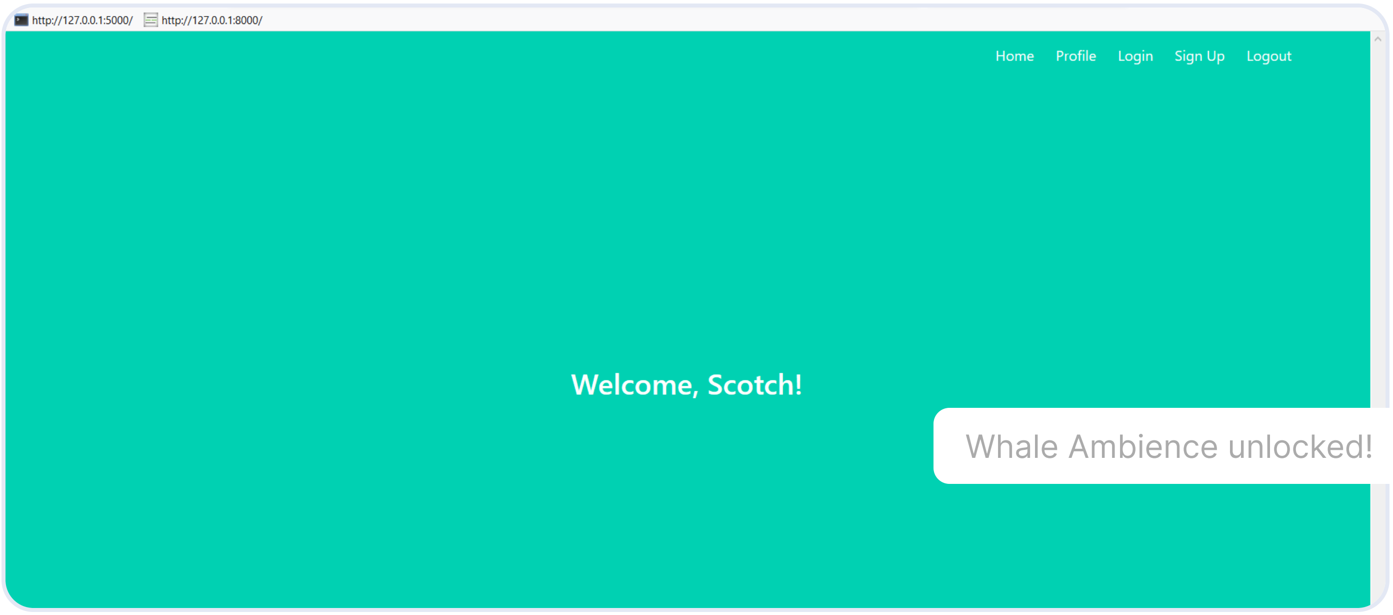
```
from flask import Blueprint, render_template
 from flask_login import login_required, current_user
from . import db
...
@main.route('/profile')
 @login_required
def profile():
    return render_template('profile.html' , name=current_user.name )
```

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 a user visits the profile page, they will be greeted by their `name`.



Whale Ambience unlocked!

Now to update the logout view, call the `logout_user` function in a route for logging out:

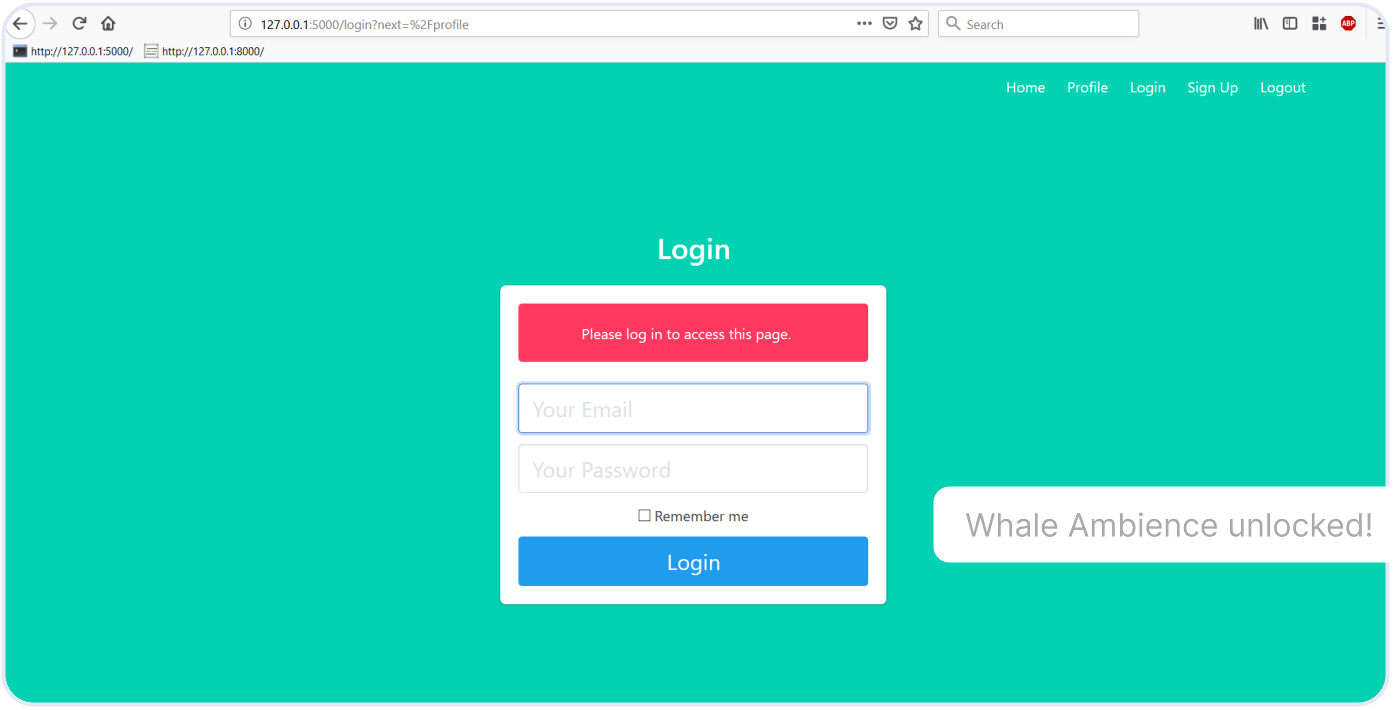
project/auth.py

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

Copy

Use the `@login_required` decorator because it does not make sense to log out a user that is not logged in to begin with.

After a user logs out and tries to view the profile page again, they will be presented with an error message:



This is because Flask-Login flashes a message when the user is not allowed to access a page.

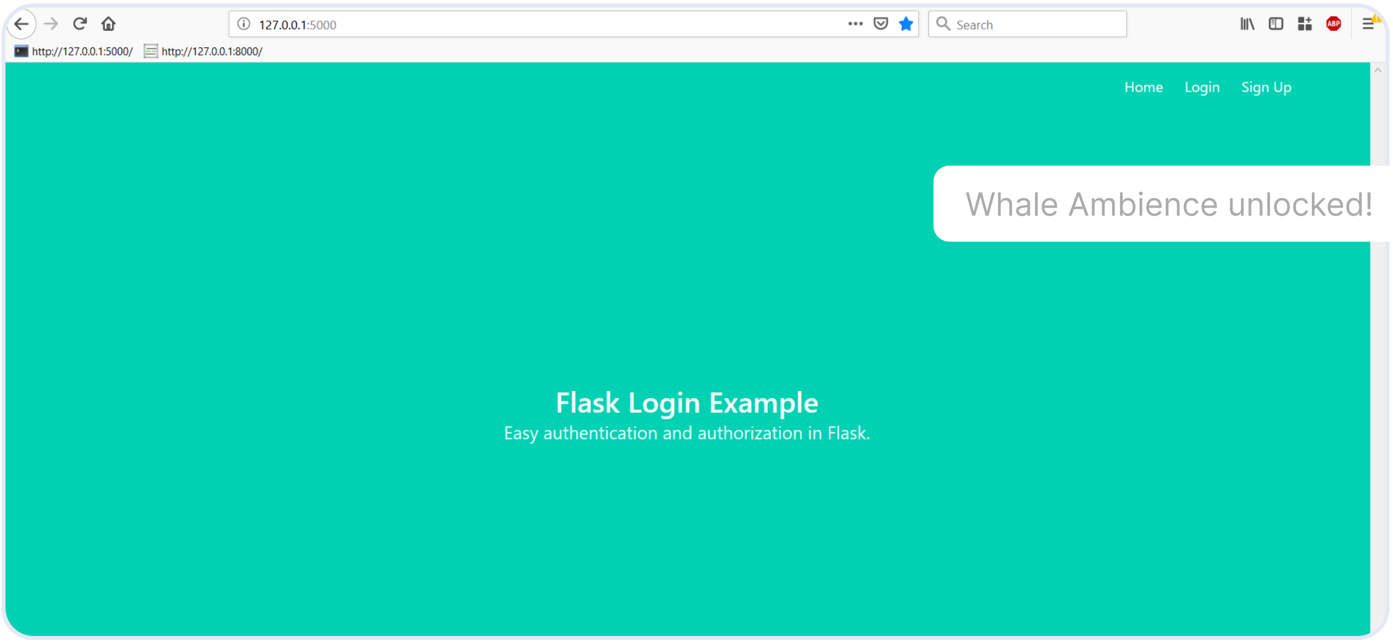
One last thing to do is put `if` statements in the templates to display only the links relevant to the user:

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

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.



Whale Ambience unlocked!

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

Conclusion

In this tutorial, you used Flask-Login and Flask-SQLAlchemy to build a login system for an app. You covered how to authenticate a user by first creating a user model and storing the user information. Then you 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, you added authorization to the app by using the `@login_required` decorator on a profile page so only logged-in users can see that page.

What you 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.

nt to deploy your application quickly? Try Cloudways, the #1 managed hosting pr r for small-to-medium businesses, agencies, and developers - for free. DigitalOcean and Cloudways together will give you a reliable, scalable, and hasslefree managed hosting experience with anytime support that makes all your hosting worries a thing of the past. Start with $100 in free credits!

Learn more here →

Whale Ambience unlocked!

About the authors

Anthony Herbert Author

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

10 Comments

Leave a comment...

This textbox defaults to using `Markdown` to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

Whale Ambience unlocked!

holla22 • November 3, 2020

For anyone struggling with the database creation. It's missing an import part for the models. I've added models at the end after create_app else won't create the user table specified.

```
>>> from project import db, create_app, models
>>> db.create_all(app=create_app())
```

Hope it helps someone getting stuck.

Show replies Reply

calsmal • March 18, 2021

Hello - wondering if anyone can help with the database configuration aspect of this tutorial.

I have followed closely but I am receiving the error 'AttributeError: can't set attribute' when initiating the db.create_all command within Python.

I have used the suggestion in the comments to import models but still no joy. Anyone able to explain where I'm going wrong?

ks!

Show replies Reply

blessedmadukoma • May 3, 2020

Please this did not work for me. From the login section, I got this error AttributeError: 'User' object has no attribute 'is_active'

Why? I followed every step but I'm getting that error. Any idea please?

Show replies Reply

Whale Ambience unlocked!

3b6bcbc6e98c47c796b3f4f9a9b620 • March 11, 2024

```
@login_manager.user_loader
def load_user(user_id):
    _# return User.query.get(int(user_id))_
    user = None
    print(f'{user_id=}')
    with Session.begin() as session:
        user = session.scalars(select(User).where(User.id == user_id)).f
        if user:
            print(f'{user=}')

            return User(email=user.email, )
```

always prints >> user_id='None'

Reply

b5188c35b9ad426ead34f9b49daf28 • January 25, 2024

[illegible] best tutorial on this topic I've ever seen. Correctly explained and it is clear wh[illegible], what and in which file changes or additions should be made. Thank you!

instead of **method='sha256'** I used **method='scrypt'** but the "password" field should then be larger, in my case: password varchar(300)

Reply

d72e9d812ef2495f892f3733b3d5e2 • December 30, 2023

How can I write a function in this backend, wich handle a post request from axios, and I can get the information about the logged in user?

Whale Ambience unlocked!

Reply

Patrick Boertje • May 17, 2023

Please update the tutorial.

The db.create_all() function is not working anymore, instead after the `db.init_app` in `__init__.py`, put the next two lines.

```
with app.app_context():
        db.create_all()
```

Reply

c75763f3917a42bb8b110804ec • October 11, 2022

Sorry, but where am I supposed to add the headers for CORS? I keep getting errors... :/

Reply

CuteSeaGreenScubaDiver • June 18, 2022

Minor error: The label on the second code block in part 7 should be changed to read `project/auth.py` instead of `auth.py`, so that it is consistent with the other code blocks for the same file. This is confusing – as it makes it seem like there is another `auth.py` file outside the `project/` directory.

Reply

Whale Ambience unlocked!

luckybokadia655 • December 13, 2021

how to set up an admin page with this project?

Reply

Load More Comments

This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.

Try DigitalOcean for free

Click below to sign up and get **$200 of credit** to try our products over 60 days!

Sign up

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

Whale Ambience unlocked!

All tutorials →

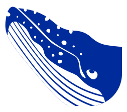
Talk to an expert →

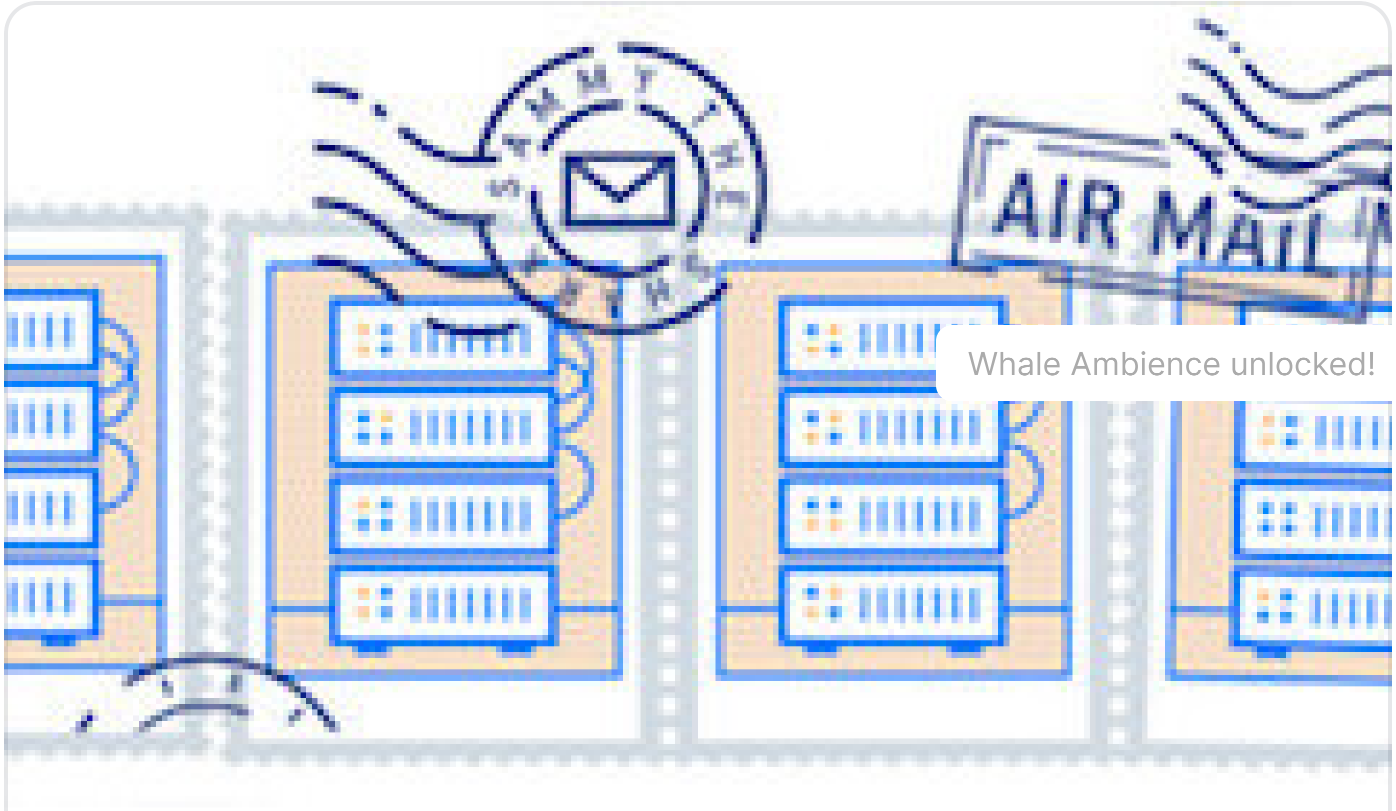
Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.

Thank you to the Glacier Bay National Park & Preserve and Merrick079 for the sounds behind this easter egg.

Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the Whale and Dolphin Conservation.

Reset easter egg to be discovered again / Permanently dismiss and hide easter egg





Get our biweekly newsletter

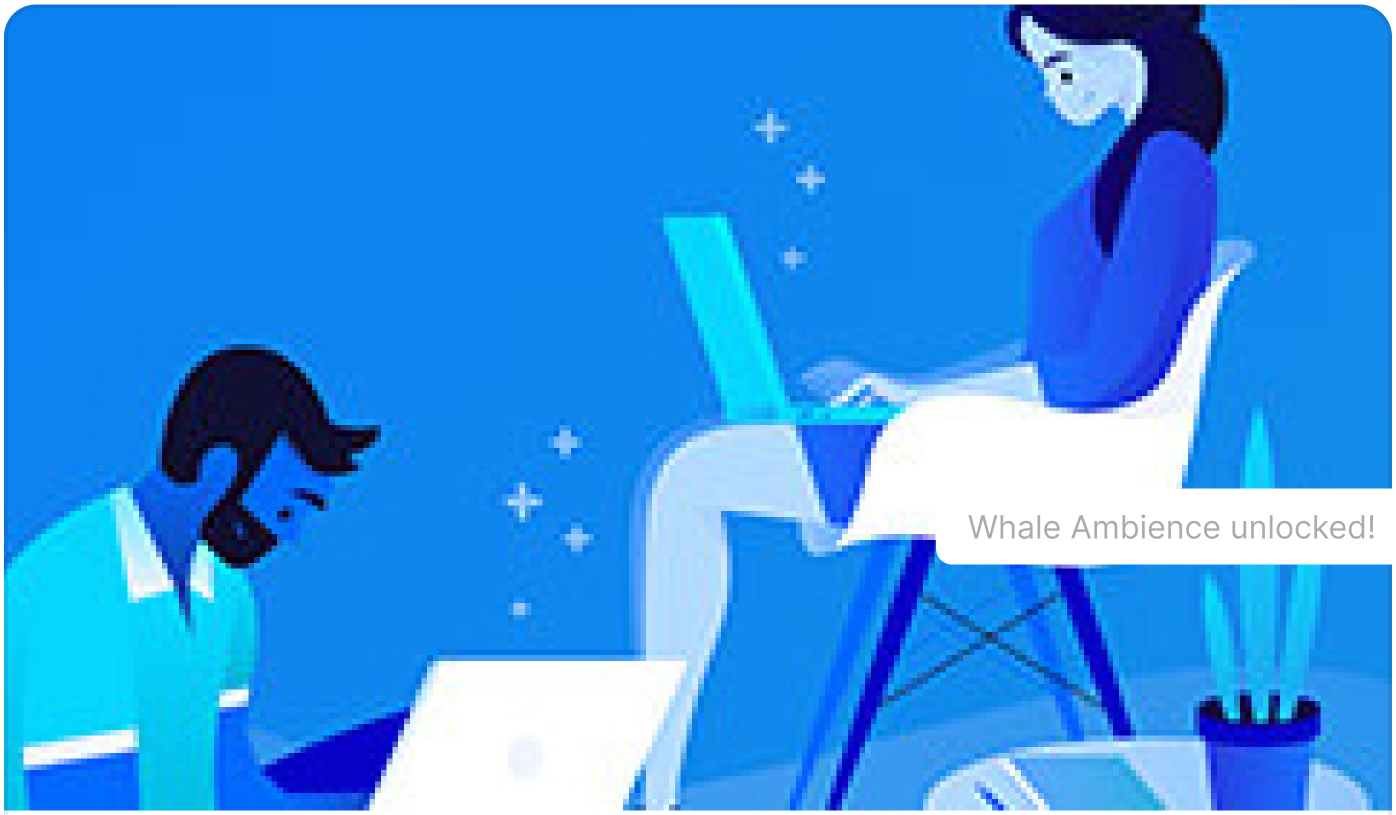
Sign up for Infrastructure as a Newsletter.

Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →



Become a contributor

Get paid to write technical tutorials and select a tech-focused charity to receive a matching donation.

Learn more →

Featured Tutorials

Kubernetes Course   Learn Python 3   Machine Learning in Python

Getting started with Go   Intro to Kubernetes

DigitalOcean Products

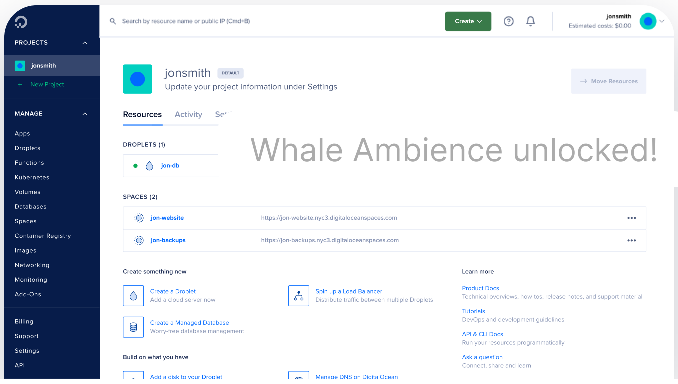
...dways   Virtual Machines   Managed Databases   Managed Kubernetes

Block Storage   Object Storage   Marketplace   VPC   Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow — whether you're running one virtual machine or ten thousand.

Learn more



Whale Ambience unlocked!

Get started for free

Sign up and get $200 in credit for your first 60 days with DigitalOcean.

Get started

This promotional offer applies to new accounts only.

Company

Products

Community

Solutions

Contact

© 2024 DigitalOcean, LLC. Sitemap. Cookie Preferences

Whale Ambience unlocked!