Task 3: Authentication and Admin Panel

Objective: To implement a complete user authentication system, including registration, login, logout, and a protected admin dashboard for approving new users. By the end of this task, a user will be able to register, but they will not be able to log in until an admin approves them.
Estimated Time: 4-6 hours

Step 1: Install Essential Libraries

Action:
You will need two more libraries to handle web forms and user sessions. Add them to your requirements.txt file and then run pip install -r requirements.txt in your terminal.

- Flask-WTF: For creating and validating web forms securely.
- Flask-Login: For managing user sessions after they log in.

Step 2: Enhance the User Model for Authentication

File to Modify: app/models.py
Action:
Your User model needs to be updated to handle password hashing and to work with the Flask-Login extension.

1. Import UserMixin from flask_login and generate_password_hash, check_password_hash from werkzeug.security.
2. Make your User class inherit from UserMixin in addition to db.Model. (e.g., class User(UserMixin, db.Model):)
3. Add two methods to your User class:
    - set_password(self, password): This method will take a plain-text password, generate a secure hash from it, and store that hash in the password_hash field.
    - check_password(self, password): This method will take a plain-text password and compare it against the stored hash, returning True if they match and False otherwise.

Why:

- UserMixin provides default implementations for the properties and methods that Flask-Login expects a user model to have (like is_authenticated, is_active, etc.).
- The werkzeug functions are the industry standard for securely hashing passwords. Hashing is a one-way process; you can't reverse it to get the original password, which is why it's so secure. We store the hash, not the password.

Documentation:

- Werkzeug Password Hashing
- Flask-Login: Your User Class

Step 3: Create Authentication Forms

New File: app/auth/forms.py
Action:

Create this new file to define your web forms using Flask-WTF.

1. Import FlaskForm from flask_wtf and various field types (StringField, PasswordField, BooleanField, SubmitField) and validators (DataRequired, Email, EqualTo, ValidationError) from wtforms.
2. Create a LoginForm class that inherits from FlaskForm with fields for username, password, a "remember me" checkbox, and a submit button.
3. Create a RegistrationForm class with fields for username, email, password, a "repeat password" field, and a submit button. Use validators to ensure the username and email are not already taken.

Why: Flask-WTF handles the tedious and critical work of rendering form fields and validating user input. It also provides built-in protection against Cross-Site Request Forgery (CSRF) attacks, a common web vulnerability.

Documentation:

- Flask-WTF Quickstart
- WTForms Validators

Step 4: Build Authentication Routes

File to Modify: app/auth/routes.py

Action:

This is where you'll write the logic to handle what happens when a user tries to register or log in.

1. Create a /register route that accepts both GET and POST requests. If it's a POST request, validate the form data, create a new User object, set its password using your set_password method, add it to the database, and redirect the user to the login page with a success message.
2. Create a /login route. On a POST request, find the user by their username. If the user exists and their password is correct (using check_password), you must also check if their is_approved status is True. If all checks pass, log the user in using Flask-Login's login_user function. If they aren't approved, show them a message telling them their account is pending approval.
3. Create a /logout route that logs the user out using logout_user and redirects to the homepage.

Why: These routes connect your forms (the view) to your database models (the data). This is the core logic of your authentication system.

Documentation:

- Flask-Login Usage

Step 5: Configure Flask-Login

File to Modify: app/__init__.py (Your Application Factory)

Action:

You need to initialize Flask-Login and tell it how to find a specific user.

1. Import the LoginManager class.
2. In your extensions.py file, create an instance: login_manager = LoginManager().

3. Back in your factory, initialize it: login_manager.init_app(app).
4. Tell the login manager which view to redirect to if a non-logged-in user tries to access a protected page: login_manager.login_view = 'auth.login'.
5. Create a user_loader function. This function takes a user ID and returns the corresponding user object from the database. This is how Flask-Login reloads the user object from the session on each request.

Why: The user_loader is the critical link between the user ID stored in the browser's session cookie and the actual user object in your database. It allows Flask-Login to provide the current_user variable in your templates and routes.

Documentation:

- Flask-Login: How it Works

Step 6: Build and Protect the Admin Dashboard

File to Modify: app/admin/routes.py

Action:

1. Import login_required from flask_login and create a custom decorator for checking admin status.
2. Create a /admin/dashboard route. Apply the @login_required decorator and your custom @admin_required decorator to it.
3. Inside this route, query the database for all users where is_approved is False. Pass this list of users to a new template.
4. Create another route, like /admin/approve/<int:user_id>, that takes a user's ID. This route should find the user, set their is_approved status to True, commit the change to the database, and redirect back to the admin dashboard.

Why: Decorators are a powerful Python feature that allows you to add functionality to an existing function. @login_required ensures only logged-in users can access a route. Your custom @admin_required decorator will add another layer, ensuring only users with the 'admin' role can see the dashboard.

Step 7: Create the HTML Templates

Action:

Create the necessary HTML files in your app/templates/ subfolders.

1. app/templates/auth/login.html and register.html: Create forms that render the fields from your LoginForm and RegistrationForm. Use Flask's "flashing" system to display messages (e.g., "Registration successful, please log in" or "Account pending approval").
2. app/templates/admin/approval_dashboard.html: Create a page that displays a table of pending users. Each row should show the username and email, along with a link or button that points to your /admin/approve/<user_id> route.

Why: These templates provide the user interface for all the backend logic you've just built. This is a large but incredibly important task. Take your time, test each route as you build it, and don't hesitate to ask if you get stuck. Good luck!