Challenges & Solutions:

1. Secure Password Hashing and Verification

One of the first challenges I faced was how to securely store and verify passwords. I explored various options and concluded that using bcrypt is the most suitable choice for this kind of application, it's widely trusted, battle-tested, and includes built-in salting and work factor configuration.

The implementation involved generating a salt and hashing the password:

salt = bcrypt.gensalt()

hashed_password = bcrypt.hashpw(password.encode(), salt).decode()

For login, I used bcrypt.checkpw() to securely verify that the input password matches the hashed one in the database.

2. Input Validation – Email & Password

It was important for me to enforce strong validation rules for both email and password inputs. I made sure to apply client-side validation for better user experience, but also backed it with thorough server-side checks to ensure data integrity and security.

For emails, I validated format correctness. For passwords, I required a minimum level of complexity to avoid weak credentials from being stored.

3. Email Verification Flow

Initially, I considered using a 6-digit code for email verification. However, I found that this approach added unnecessary complexity both in terms of frontend handling and token management.

Instead, I switched to a cleaner approach using a UUID-based token embedded in a verification link. This solution was easier to implement, more secure, and provided a

smoother user experience. It also allowed me to render a simple HTML page (verify_success.html) upon successful verification.

4. Implementing Rate Limiting on Login Attempts

Another important security feature was login rate limiting. At first, it didn't work as expected due to circular import issues between app.py and routes.py.

To resolve this, I modularized the Flask app structure by creating an extensions.py file that centralized configuration for shared components such as limiter, db, and jwt. This allowed the rate limiter to be properly initialized and imported without cyclic dependencies.

5. JWT Blacklist Implementation

During the logout flow, I needed a way to invalidate tokens to prevent reuse. I implemented a simple in-memory blacklist using a Python set, where the jti (JWT token ID) of each logged-out token is stored. This was later checked on protected routes to reject requests with blacklisted tokens.