Group 8

Burgin Luker, Tiegan Cozzie, Josh Wilcox, Morgan Hanson

Blabber

- Sign in and Sign up functionality with persistent login
- View messages posted by other users in chronological order
- Write your own messages to post to the message board
- Edit your username and profile icon

Github: https://github.com/Tcozzie/Babble

Slides: https://github.com/Tcozzie/Babble/blob/main/331%20Final.pdf

Creative Objective:

When brainstorming ideas for our project, through thoughtful, incessant, deliberations we realized an app that allowed the members of our class to communicate with one another in a fun user friendly manner would be an outstanding final project. Using this idea as our basis, we figured simplicity of use, availability, and minimal user interface related clutter would be integral features of our app. Moreover, we figured what users would value most is being able to quickly post sentences amongst each other, thus the idea for our simple twitter-esque application was born.

Tech Summary:

- Sign In / Sign Up
    - This functionality was built using the Amazon Cognito API paired with our front and back end code. The Cognito API provides secure, frictionless customer identity that is easily scalable and handles valid email address checking, password verification and storage, along with a user-friendly console that allows us to see who has created accounts on your app. Henceforth, as programmers, we only needed to create a simple form screen that allowed a user to sign up or sign in with the information we requested and upon submission handle that request on our backend by invoking the appropriate Amazon functions.
- Front End Functionality

- All visual features such as, viewing messages, creating messages, editing profile icon, and viewing of loading indicators were done using a mixture of html, htmx, and jinja templates. Beginning with Jinja, this templating engine allowed us to create simple html with variable content that was portable and reusable throughout our app, for instance, one message has its own template and can be used in multiple places. Additionally, the Python-like syntax and functionality allowed the use of loops and conditional statements in order to change what was rendered for the user. Furthermore, htmx was a quintessential aspect of our project, as it provided easy functionality for scrolling through our list of messages and performing hx-swap on the end of the list to keep it growing. As well, htmx allows for DOM manipulation within each message template, thus allowing us to change the like amount of each message. On top of that, while loading the messages, htmx provided us with a built in, customizable loading indicator that we use while loading the messages.

- Back End Functionality:
    - All CRUD operations regarding users and their messages were performed on our back end which consisted of Flask, SQLite, Peewee ORM, and Redis. Flask is a lightweight web application framework that is used to handle web routing logic. Throughout our project we used Flask to issue either a GET or POST to either send data to our backend end code that would create a message, request a list or message, or any other task we

needed. Regarding data storage, we elected to use SQLite, which provided us with a simple way to store all data related to either users or messages on the app. Building upon SQLite, we utilized Peewee ORM in conjunction with our database so that we did not have to spend time writing SQL and creating tables, rather we could use the API of the object relational model in order to quickly and simply perform database interactions for us. Finally, Redis was used as a quicker way to save the data about message likes and which user liked them. It was somewhat difficult storing an array of users that had liked each message within the message itself, therefore, we elected to use the speed and concurrency Redis offers to easily store the amount of likes on a message and who had liked it already.

Member Notes:

- Burgin Luker
    - Back end file structure, database design, routes and accompanying code for creating a message, getting messages, the Python model classes such as user and message. Additionally, front end features such as making the Jinja template for listing messages, the htmx for infinite scrolling and loading indicators, and showing the user profile page. Overall my work involved Jinja template creating, html/htmx, and writing a lot of Python code using both the Flask API and the Peewee API in addition to writing

the code to handle the logic for handling message interaction in our database.

    - One thing I learned is how valuable it is to read documentation and to be able to understand documentation quickly. For instance, people that make products such as Peewee and Flask want their products to be easy to use, therefore, read what they wrote for you and it will be unquantifiably helpful. In hindsight, I would have picked up on this idea sooner. For example, I fought plenty of python errors due to mistakes I was making with the Peewee, get() function that is meant to get a row from the database. This function, if no row matches what one is attempting to get, will error. However, had I been more vigilant in my documentation studying, I would have discovered get_or_none(), which does not error, much sooner. This is only one example of my novice developer mistakes and is indicative of a broader idea I learned during this project which is: If something feels silly, it probably is, so look for a better way.

- Tiegan Cozzie
    - Implemented routes and backend logic for creating messages, retrieving messages, comments, and likes, along with model classes for date management and ordering. Designed and styled various frontend features, such as profile icons (and more), while handling user authentication end-to-end with Amazon Cognito. This included building sign-up/sign-in forms on the frontend and integrated Amazon Cognito on the backend for tasks like managing cookies, user permissions, and XXS protections.

Managed all DevOps for the project. Registered the domain **b4bble.com** via Cloudflare, configured HTTPS enforcement, and implemented end-to-end encryption with SSL certificates. Deployed the application on an Amazon EC2 instance and secured it to only accept connections from Cloudflare, with shared public/private encryption keys. The EC2 instance is running NGINX which listens to port 443 and port 80. Re-routes all traffic to 443 and serves as a reverse proxy to then send requests to port 8000 (the port the server is running on). Gunicorn is wrapped around the Flask server as flask isn't made for production. Gunicorn is running two processes in parallel to be able to handle 2x the requests to the server.

    - One important lesson I learned is the value of writing secure code. For example, if traffic is transmitted over HTTP instead of HTTPS, and cookies are not signed, an attacker could intercept unencrypted traffic using tools like Wireshark or a WiFi Pineapple. This allows them to steal cookies, which are often used for user authentication, and impersonate users across the website. While enforcing HTTPS is crucial to encrypt traffic and prevent such attacks, signing cookies adds an additional layer of protection. In situations where HTTP might still be used, a signed cookie ensures that even if intercepted, the attacker cannot tamper with or forge the cookie without the private key used for signing.
- Josh Wilcox
    - Created the feature to delete and edit messages. The Frontend for this feature is an options button that displays a menu containing the edit and

delete option when clicked. Selecting the edit button triggers a route that targets the id of the paragraph containing the message, replaces it with html located in another folder, and when the user submits the edited message, a backend route returns html with the updated message. The delete route is much less complicated. Once the user selects the delete button an hx-delete request is issued, and the message is removed within a route using a simple peewee function call, as well as that message's likes and comments.

    - This project gave me more experience learning to style frontend features, and utilize htmx's functionality. I also learned the importance of compartmentalizing your html templates to make the frontend code easier to read. A few things I would have liked to do given more time would be to properly implement comment threads, similar to how reddit handles comment threads. This would introduce some cool new problems that would require recursion. Also, adding the ability to bookmark messages from other users would have been a cool idea as well.

- Morgan Hanson
    - Primarily focused on front-end development by creating a seamless and interactive user experience. A key part of my work was implementing global styling across the app to ensure consistency in design and layout across all pages. Used Jinja to create a reusable tweet template that was used across multiple different pages throughout the site. This template allowed for dynamic content rendering that integrated seamlessly with the

backend code. I also implemented the character count functionality for both creating posts and posting comments. This feature was built using on input events which facilitated real-time updates while the user types. This dynamic functionality restricted the user from typing over the character limit. Overall, my work included HTML/HTMX, CSS, JavaScript, and Jinja.

- This project allowed me to explore HTMX and Jinja as I had no previous experience with these technologies. They taught me the value of dynamic website interactions, where content is updated without full-page reloads which allows for a greater user experience and site functionality. I also learned the importance of modular front-end design. By creating reusable templates, I was able to reduce redundancy within the code while also maintaining consistent UI across different pages of the site. This approach taught me how to streamline the development process, leading to more scalable and maintainable code.

Conclusion:

As a group, one major takeaway we have from this project is the importance of compartmentalization throughout collaboration. This project was wonderfully crafted for each developer to possess specific tasks, for instance, using templates allowed each of us to to develop templates along with specific backend routes separately, however, in the end we could merge them together in one cohesive product. Regarding decisions that worked, using an object relation model, as well as using git for source control, were

fantastic decisions. For example, most database interactions were quite simple, therefore, the ORM significantly expedited CRUD processes while also deflating the length of the codebase by preventing unnecessary SQL strings. Additionally, using git allowed us all to participate without needing to meet and allowed easy access to each version of the project throughout development. One thing that did not work well was trying to use a css library, pico.css, in order to streamline front end styling. This did not work because it made customization of our front end templates overall more difficult because we found ourselves constantly having to overall default pico styling. Thus, it would have been much easier to simply begin by styling things ourselves. Overall, we would have done almost everything the same, however, we would have considered using a hosted database rather than a local file system database. This is because database migrations were difficult and git does a poor job tracking .db files. This caused multiple idiosyncrasies between developers with different db versions. These issues could have been avoided with a hosted database in which we all had admin privileges.

References

Authentication Service - Customer Iam (CIAM) - Amazon Cognito - AWS, aws.amazon.com/cognito/. Accessed 22 Nov. 2024.

"HX-Indicator." *Htmx ~ Hx-Indicator Attribute*, htmx.org/attributes/hx-indicator/. Accessed 22 Nov. 2024.

"Jinja¶." *Jinja*, jinja.palletsprojects.com/en/stable/. Accessed 22 Nov. 2024.

"Sqlite Home Page." *SQLite Home Page*, www.sqlite.org/. Accessed 22 Nov. 2024.

"Welcome to Flask¶." *Welcome to Flask - Flask Documentation (3.1.x)*, flask.palletsprojects.com/en/stable/. Accessed 22 Nov. 2024.