Wilson Quilli

Professor Nalubandhu

SWENG 861: Software Construction

January 31st, 2026

Wilson's Week 3 Assignment Backend Report

The application I created is a social network for college students, where they can create profiles and document/share how they're doing in school and life. Only college students can create accounts, and an account will expire automatically after the user graduates. This ensures that the service will always be a student community only. In this assignment, I experimented with using APIs and began my final project by utilizing the Dog API to show dog breeds and sub-breeds. The backend was created using Flask/Python for both user account information, as well as storing the breed/sub-breed data in an SQLite database. Additionally, the application has Google as the social login provider using Option A: Social Login (B2C), which uses OAuth 2.0/OpenID Connect as the underlying authentication method. Therefore, the application provides users with a secure, familiar way to log into the application and, at the same time, eliminates the risk and cost of developing custom authentication.

When a user selects to sign in using Google, they will be redirected to Google's OAuth authorization server where they will need to authorize the application. When a user is successfully authenticated, the Google server will redirect the user back to the application with an authorization code. The application will then exchange that code with Google for credentials. After receiving the tokens, the application will validate the OpenID Connect ID token, and

retrieve the user's identity and attributes, including email, name, etc. At this stage, the application will either create a new user in the database, or update the corresponding record for a matching user. Upon successful login, the application will return a JSON Web Token to the user. The JWT allows the user to access the OAuth's protected API endpoints. To enforce authentication for the protected API endpoints, the application will check the JWT against its internal validation rules, check for user roles, and apply rate-limiting rules per a user's IP to prohibit a user from abusing one of these protected API endpoints from the same IP address. The protected API endpoints will enforce administrative operations only to authorized users, while regular users will only be allowed to access the endpoints designated for their user roles.

One of the important aspects of security during the development process was how the application relied on an authenticated session context instead of object-level authorization to ensure that the risk of data being accessed by unauthorized users was reduced. This meant that there was no sensitive information available in an API response, and the error messages provided to the client were cleaned up so that internal system details would not be disclosed to the client as a result of an API error response. Other features implemented for security include rate limiting and logging. Overly frequent requests from any single IP address would be met with a 429 response to indicate "Too Many Requests." Both an authenticated user experience and an authenticated user experience were tested by performing manual tests in both Chrome and Postman, and demonstrated that security features were in place.

The application also has RESTful API endpoints. The GET /api/hello endpoint returns a personalized greeting for authenticated users and a 401 Unauthorized response for unauthenticated requests. The application integrates the Dog CEO API to retrieve a list of dog

breeds and sub-breeds, which are stored in a local SQLite database. Validation ensures that all breeds are non-empty strings and duplicate entries are ignored. Admin users can fetch external breeds using the POST /api/breeds/fetch_external endpoint. This process updates the local database while clearing the cache of previously stored breeds to ensure consistent data retrieval.

To meet the full requirements of a RESTful API, the application implements complete CRUD operations for both breeds and sub-breeds. Admin users can create, update, and delete breeds and sub-breeds through dedicated endpoints. Each operation includes validation, proper HTTP status codes, and appropriate error handling, returning 400 for invalid input, 404 for missing resources, 201 for creation, and 200 for successful updates or deletions. Deleting a breed automatically removes its associated sub-breeds, to maintain referential integrity in the database. These endpoints stick to RESTful principles, using consistent URL structures, standard HTTP methods, and meaningful responses, ensuring clarity and maintainability.

The database architecture consists of two main components: users.db for user account storage, including ID, email, role, and timestamps, and dog_api.db for breeds and sub-breeds. Sub-breeds are linked to their parent breeds through foreign keys, and the database is initialized at startup to create necessary tables if they do not exist. Caching is applied to breed queries to optimize performance, and caches are cleared whenever data changes. Logging and rate-limiting mechanisms complement this architecture, providing a balance between performance, security, and usability.

Overall, the application successfully meets its objectives of providing a secure yet user-friendly social media platform for future students and their friends. The use of APIs and security methods such as Google OAuth, JWT Web Token-based authentication with multiple

roles, as well as full CRUD design and validation for all breeds/sub-breeds displayed through RESTful API, demonstrates that the application has been designed correctly. To summarize, the assignment establishes a fully functional and secured backend, showing that it is possible to do so before developing the actual social media application for students.