BlogHub : Designing a API for a scalable Multi-User Blogging platform

Seul Lee
Department of Computer Science
University of Auckland
Auckland, New Zealand
else568@aucklanduni.ac.nz

***Abstract*—This report explains the overall structure of a multi-user blogging system, including the backend, frontend and database. In particular, I describe my contributions to both backend and frontend development, focusing on the API design and component structure. The report also provides an overview of how components interact within the frontend architecture.**

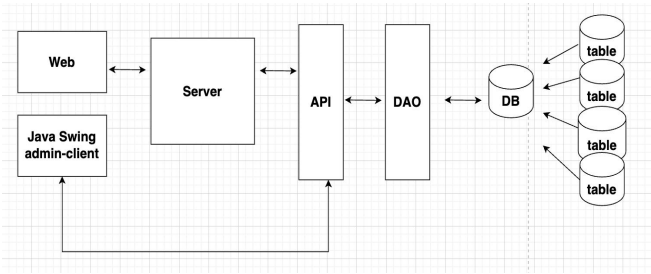
Keywords— backend development, SvelteKit, Node.js, SQLite

I. Introduction

When we started our project, we discussed the overall structure. We created the ER diagram and API list together. Our project's overall structure is as follows: we split the roles into backend, database, frontend and Java Swing. For the frontend, we used the Svelte stack. For the backend, we implemented a Node.js and Express API. In Java Swing, we handled login authentication, fetched the user list and loaded avatar images by communicating with the backend via HTTP.

II. Overall System Structure

A. System Flow Diagram



When a user sends a request from the web frontend, such as posting a new article or delete one, it is delivered to the backend as an HTTP request (e.g.,POST/articles, DELETE/articles/:id). The Express server receives the request and routes it to the appropriate API endpoint.

Within the API layer, the route handler calls a corresponding function from the DAO. For example, the function deleteComment() is called when a request to delete a comment is received.

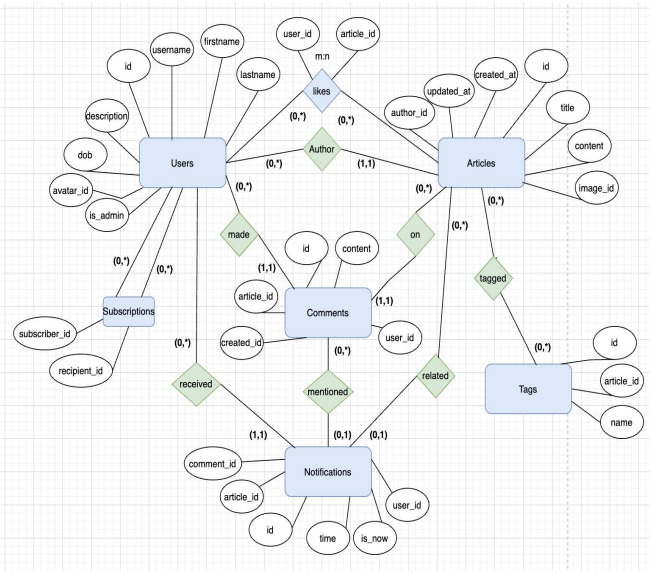
Inside the DAO, we use a utility function getDatabase() from database-dao.js, which opens a connection to the SQLite database. If the database file does not exist, it will initialize SQL script defined in an environment variable.

After the database connection is ready, the SQL query is executed. Depending on the request, one or more tables are queried or updated, such as Users, Articles or Comments.

The result of query is then returned through the DAO, passed through the API part, and sent from the Express server to the frontend in JSON format.

The Java Swing admin client works in a similar way as the web frontend. It only communicates with the Express API server using HTTP and never directly accesses the database. Functions like user login, retrieving the user list, and displaying avatars are all handled through API calls.

B. Database Diagram



The database is designed to support various blog functionalities and includes several interconnected tables: Users, Articles, Comments, Tags, Likes, Subscriptions and Notifications. Each table plays a specific role, and they are connected through foreign key constraints to reflect real-world relationships between data entities.

The Users table stores basic information about each user, such as their username, date of birth etc. And whether they have admin privileges. This is the core entity, as most other tables reference users in some way. For example, the Articles table contains posts written by users and includes fields like

title, content, created_at and author_id, which is a foreign key linking back to the Users table. This shows which user authored a specific article.

Each article can have multiple comments, and this is represented by the Comments table. A comment belongs to one article and is made by one user. Therefore, the Comments table includes two foreign keys: article_id, and user_id, pointing to the Articles and Users tables, respectively. This allows the system to retrieve all comments for a specific article and show who posted them.

Articles can also be tagged with keywords using the Tags table. Each tag is linked to one article via the article_id foreign key. This helps categorize articles and allows users to filter or search content based on tags.

The Likes table records which users have liked which articles. It represents a many-to-many relationship between users and articles, so it contains two foreign keys: user_id and article_id. These foreign keys reference the Users and Articles tables. This design enables features like showing how many likes an article has received or whether a specific user has already liked it.

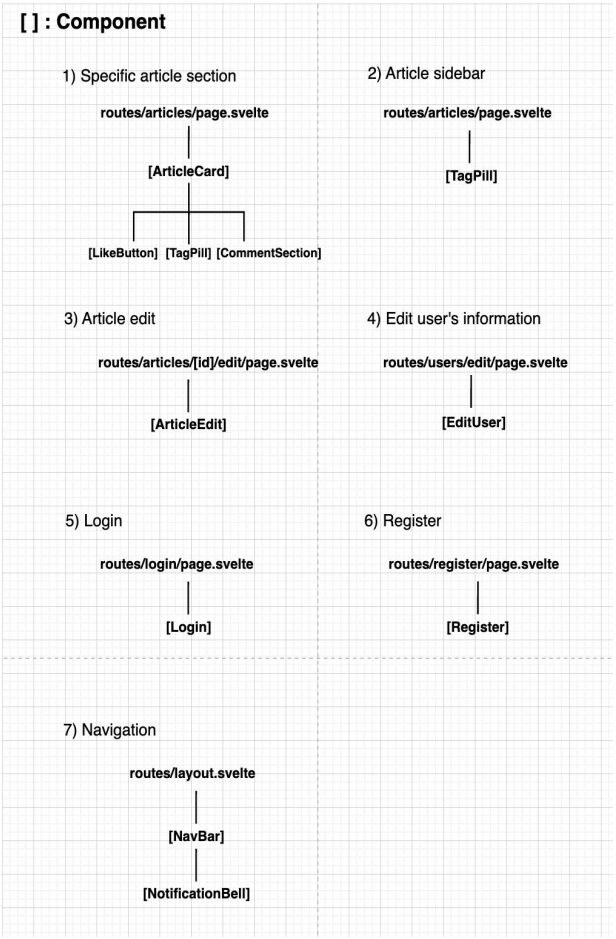
The Subscriptions table manages user-to-user follow relationships. A subscription means that one user follows another user. This table has a composite primary key made up of subscriber_id and recipient_id, both of which are foreign keys referencing the Users table. A CHECK constraint ensures that a user can't subscribe to themselves. This table supports features like showing follower list or generating a feed based on the users someone follows.

The Notifications table is used to alert users when certain actions occur, such as when they are mentioned in a comment. Each notification belongs to one user and may optionally reference a specific article or comment. To support different types of notifications, the Notifications table includes nullable fields article_id and comment_id. Depending on which field is not null, the application can determine whether the notification is related to an article or a comments. These fields are foreign keys that link to the Articles and Comments tables.

All these relationships are enforced using SQL foreign key constraints which ensure referential integrity between tables. This means that, for example, a comment can't exist without a valid article or user and a like can't exist without a corresponding user and article.

When a user interacts with the application such as liking an article, posting a comment or following another user, the frontend sends a request to the backend. When the request approach database, Those queries might insert, update or retrieve data from multiple related tables.

C. Component Structure Diagram



The frontend is composed of ten components, each of which is functionally distinct yet interdependent. These components are organized by feature and their dependencies are structured to support modular development and maintainability.

The routes/articles/page.svelte file displays a specific article and manages related interactions. It renders the ArticleCard component, which in turn depends on three subcomponents: LikeButton, TagPill and CommentSection. These subcomponents receive data and event handlers from ArticleCard. For example, LikeButton handles user reactions, TagPill manages tag filtering and CommentSection handles comment creation and deletion. This modular structure allows for reusable and isolated functionality within the article view.

The same TagPill component is also used in the article sidebar for filtering articles by tag. It receives props like tag and selected and emits an event when clicked to toggle tag selection.

The article editing page renders the ArticleEdit component. This component receives the articleId as a prop and allows the user to modify an existing article.

The edit page about user's information renders the EditUser component, which receives the currently logged-in user as a prop. It allows the user to update their profile information.

The login page and register page render the Login and Register components, respectively. These pages handle user authentication and account creation.

Finally, the layout file provides the global navigation structure of the application by rendering the NavBar component and passing in the number of unread notifications. It uses a shared store(userStore) to track the current authentication state and set it based on the received data loggedInUser. The NavBar component uses the userStore to determine whether the user is logged in and conditionally displays different navigation options. If a user is logged in, it shows links like Dashboard, Subscriptions and New Post, as well as the NotificationBell component and avatar button. If not, it shows Login and Join buttons. The layout and navigation are dynamically reactive, ensuring that the UI updates as the user's login status changes.

III. My Contributions

My contributions can be divided into four parts. First, I created a figma design document to help my team members understand the overall structure of the homepage before we officially started the project. Second, I designed most of the API structure and DAO, excluding the notification, subscription, upload and login features. Although I was not directly responsible for these parts, I will briefly explain them to provide a comprehensive understanding of the overall API design. Third, I implemented the tag, like, search and sort functionalities on the frontend, covering requirements 9, 11 and 14. Lastly, I fixed and improved the functionality related to article editing and subscription features.

A. Figma design document

I created a Figma design document that outlines the entire homepage interface. It consists of thirteen frames, including login, registration, article list pages(with sort, search and tag features) and detailed article pages(with subscription, likes, comments, tags and notifications).

Each interface was designed based on the requirement document.

In addition, I visually represented the functional differences between logged-in and non-logged-in users-such as the ability to write articles or post comments-to help team members clearly understand the overall frontend structure.

B. Api structure

1. Articles API
   - GET /api/articles
     - Description : Retrieve all published articles with their associated tags and comments (for non-logged-in users as well).
     - DAO : getAllArticles()
   - GET /api/articles/:id
     - Description : Retrieve a specific article by ID along with its comments, likes, tags, and whether the current user has liked it.
     - DAO :getArticleById(id),getCommentsByArticleId(id), countLikes(id), getTagsByArticleId(id), inLiked(id, userId)
   - POST /api/articles
     - Description: Create a new article(authentication required).
     - Validation:
       - Uses Yup schema:
         - {title : string (required), content: string(required), tags: array of single-word strings}
     - DAO : createArticle(articleData)
   - PATCH /api/articles/:id
     - Description: Update an article(authentication and ownership required).
     - Validation : Same Yup schema as in POST used to validate updates.
     - DAO : updateArticle(id, userId, updatedData)
   - DELETE /api/articles/:id
     - Description: Delete an article(authentication and ownership required).
     - DAO : deleteArticle(id, userId)
   - GET /api/articles/tags
     - Description : Get all tags used in articles.
     - DAO : getAllTags()
   - GET /api/articles/tags/:tag
     - Description : GET all articles that contain the given tag.
     - DAO : getArticleByTag(tag)

2. Comments API
   - POST /api/comments
     - Description : Create a new comment for a specific article. Mentions(e.g., @username) in the comment content trigger notifications to mentioned users. Requires authentication.
     - DAO : createCommentWithMentions({content, articleId, userId})
   - GET /api/comments/:commentId
     - Description : Retrieve a specific comment by its ID, including the commenter's username.
     - DAO : getCommentById(commentId)
   - DELETE /api/comments/:id
     - Description : Delete a comment. Only the comment's author, the article's author, or an admin can delete. Requires authentication.
     - DAO : deleteComment(commentId, userId)
   - GET /api/comments/articles/:articleId
     - Description: Retrieve all comments associated with a specific article, ordered by creation time(ascending).
     - DAO : getCommentsByArticleId(articleId)

3. Likes API

- POST /api/likes
  - Description : Toggle like(like or unlike) for an article by the logged-in user. Requires authentication.
  - Request Body: {articleId}
  - DAO : toggleLike(articleId, userId)
  - Response : {message: “Like toggled”, liked: true | false}
- GET /api/likes/:articleId/count
  - Description : Retrieve the number of likes for a specific article.
  - DAO : countLikes(articleId)
  - Response: {likes : number}
- GET /api/likes/:articleId/status
  - Description : Check whether the current user liked a specific article. Requires authentication.
  - DAO : isLiked(articleId, userId)
  - Response : {liked: true | false}

4.Users API

- GET /api/users/me
  - Description : Get the currently logged-in user’s profile with their articles. Requires authentication.
  - DAO : getArticlesByUserId(userId)
  - Response : {user: {…userInfo, articles: […]}}
- GET /api/users/:id/articles
  - Description : GET all articles written by a specific user.
  - DAO : getArticleByUserId(userId)
  - Response : {articles: […]}
- GET /api/users
  - Description : Get a public list of all users.
  - DAO : getUsersPublic()
  - Response : [{id, username, firstname, lastname, avatar_id, is_admin}, …]
- PATCH /api/users/me
  - Description : Update profile of the current user. Supports partial update.Requires authentication.
  - DAO : updateUser(userId, updateData)
  - Response : 204 no Content if success, 422 on validation error
- GET /api/users/:id
  - Description : Get public profile info of a user by ID.
  - DAO : getUserById(userId)
  - Response : {username, firstname, lastname, avatar, ..}
- DELETE /api/users/:id
  - Description : Delete a user and all their related data. Requires authentication(user must be self or admin).
  - DAO : deleteUserAndRelatedData(userId)
  - Response : 200 OK or 403 Forbidden or 404 Not Found

5.Admin API

- POST /api/admin/set-admin-password
  - Description : Promote a user to admin and set a new password. Requires login authentication.
  - Body : {username, newPassword}
  - DB : UPDATE Users SET password = ?, is_admin = 1 WHERE username = ?
  - Response : {message : “Admin password set successfully”} or 404
- GET /api/admin/users
  - Description: Get all users with article count. Admin-only. Requires admin authentication
  - DAO : getAllUsersWithArticleCount()
- DELETE /api/admin/users/:id
  - Description : Delete a user and all related data. Admin-only. Requires admin authentication
  - DAO : deleteUserAndRelatedData(userId)
  - Steps:
    - BEGIN TRANSACTION
    - Delete notifications(by comment, article, user)
    - Delete likes(by user & their articles)
    - Delete subscriptions
    - Delete comments
    - Delete tags(from their articles)
    - Delete user’s articles
    - Delete the user
    - COMMIT or ROLLBACK on error
  - Response : 204 No Content or 500 Internal Server Error

6.User Authentication API

- The authentication system includes login, logout, and user registration via JWT-based token handling. The requiresAuthentication middleware ensures that only authenticated users can access protected routes by validating the JWT stored in cookies. Additionally, user-related data access operations, such as registration, login verification using bcrypt, and profile updates with Yup validation, are handled in the users-dao.js file.

7.Subscription API

- The subscription system allows users to follow or unfollow other users. It provides endpoints to check subscription status, subscribe or unsubscribe from a user, and retrieve subscriber/following lists. The subs-dao.js file handles the database logic, while the route uses authentication middleware to ensure only logged-in users can perform subscription actions.

8.Notification API

- The notification feature informs users about relevant events, such as new comments or articles from subscribed authors. Authenticated users can fetch their notifications, and mark them as seen. Notifications are automatically created such as when a user leaves a comment.

9.File Upload API

- The image upload feature allows authenticated users to upload image files via a /uploads endpoint. The server renames the file with a unique ID and moves it to the public directory for access. While I didn't develop this module, it supports avatar and content image management across the platform.

C. Implement the frontend functions

1. tag function

I implemented the tag filtering feature on the list page to meet the requirement of allowing users to filter articles by tag. The page first retrieves and displays the full list of available tags at the top of the article feed. Each tag is rendered using a reusable Tag component, which receives three props: the tag name, a selected boolean indicating whether it is currently active, and an onClick handler.

When a tag is clicked, the component toggles its selection state by comparing the clicked tag to the currently selected tag. If the clicked tag is already selected, it clears the filter and shows all articles again. Otherwise, it sets the selected tag and filters the articles to only show those that contain the chosen tag.

I also implemented input validation of the article tags to ensure tag is a single word, as required. Users can enter multiple tags separated by commas, and before the article is submitted or updated, the input is validated using a regular expression to confirm that each tag contains only alphanumeric characters and no spaces or special symbols. If any tag fails this check, the submission is blocked and an alert is shown. This validation ensures the consistency of tags stored in the database and prevents malformed input from being processed by the backend.

2. like function

I implemented the like functionality by combining frontend interactivity with backend API communication. Each article displays a like button component that reflects the current user's like status and total like count.

On the frontend, when a user clicks the like button, the handleLike() function in ArticleCard.svelte is triggered. This function sends a POST request to the backend API(/likes) with the article ID to toggle the like status. Upon successful response, it updates the local state(isLiked) and fetches the updated like count using the getLike() function. A short animation(isLikedAnimating) provides visual feedback to the user.

In addition, when the component is mounted, the checkLikedStatus() function sends a GET request to /likes/:articleId/status to check whether the currently logged-in user has already liked the article. This ensures that the heart icon(filled or outlined) accurately represents the user's interaction history.

The total number of likes is fetched from /likes/:articleId/count, and the value is dynamically updated to reflect real-time changes. To prevent unauthorized interactions, the like button is disabled for the article owner and users who are not logged in are prompted to log in before liking.

This feature satisfies requirement 11, allowing authenticated users to like or unlike any article they do not own, while keeping the UI and server state synchronized.

3. search & sort function

To meet the requirements for search and sorting functionalities in the article listing page, I implemented filtering, searching and sorting logic directly in article/page.svelte.

A <select> element allows users to sort the articles by title, username or date. The selected option is bound to the sortOption variable and a custom .sort() function reorders the article list accordingly.

For the search functionality, I added an input field that allows users to search article titles, content, author username and published dates. The search is case-insensitive and includes a match mode toggle that lets users choose between partial and exact word matches. The match mode is controlled via two radio buttons bound to a matchMod variable.

- In partial mode, a keyword like "heal" will match any article with words like "partial" or "health".
- In exact mode, only exact word matches like "health" will be returned.

To support cleaner search logic, I implemented a custom sanitize() function to remove punctuation and normalize the text before comparison. This ensures that searches such as "2025's" will still work as expected.

D. Improve functions

To improve the article display and subscription features, I made the following changes:

For better readability of list items in articles, I applied custom styles to ol and ul elements using global CSS. This adds appropriate indentation and spacing to list items within article content.

To implement the subscribe functionality on article cards, I created a new store file(subscription-store.js) using Svelte's writable store. Each card checks that current subscription status of the article's author, and users can toggle this status by clicking a button. The toggleSubscription function sends a POST or DELETE request to the API depending on whether the user is subscribing or unsubscribing. The result updates the shared store and adjusts the subscriber count in real-time.

IV. Applying Concepts to the Project

For the API design, I followed the principles covered Module 5.2. I created separate route files for each

resource(e.g., users, comments, etc.) and imported them into app.js, where I used app.use to register the routes. In line with RESTful conventions, all API paths are structured using nouns to represent resources.

CRUD operations were implemented appropriately for each route. I used HTTP methods like GET, PATCH and DELETE to handle retrieving, updating, and deleting data, following REST guidelines.

In addition, Module 5.1 introduced tools like Postman and the concept of route handlers. I applied these skills to create and test API endpoints efficiently. Writing clear route handlers for different API functions allowed me to quickly test functionality using Postman, which made debugging and iteration much faster during development.

In Module 7.2 on database design, I learned how to use the JOIN keyword to combine data from multiple tables. I was able to apply this concept directly when writing backend DAO functions. For example, in the article DAO, I joined the Articles and Users tables on the author_id field to retrieve article data along with the full name of the author. This allowed me to return both the article content and related user information in a single query, making the API response more efficient and complete.

In the frontend, I implemented features such as tagging and liking using custom functions. When building more complex pages, I applied the component-based design approach covered in Module 4.1, passing props between components to keep the code modular, reusable and easier to maintain. This helped improve both the clarity and scalability of the application.

In Module 4.2, I learned how to use Svelte-specific templating features such as each blocks and if conditional rendering. These were particularly useful for dynamically displaying lists of tags and conditionally rendering UI elements based on user interactions or state.

In addition to what was covered in class, I applied transaction handeling using BEGIN, COMMIT, and ROLLBACK statements when creating articles. This ensures that all related operations-such as inserting article content, tags and notifications- are treated as a single atomic unit. If any step fails, the entire process is rolled back to maintain data integrity. This approach was not taught in class, but I found it essential for preventing partial or inconsistent database updates.

V. Conclusion : Lessons Learned from Team Collaboration

Through this project, I experienced for the first time what it's like to collaborate with others to build a shared outcome. Unlike the individual lab exercises where I wrote and tested code on my own, I initially found it challenging to read and understand code written by others. Each teammate had a different coding style, and it took some time to adjust.

However, this diversity turned out to be a valuable learning experience. I was exposed to alternative coding approaches and techniques I wouldn't have encountered if I had worked alone. For example, when implementing a certain feature, I learned that there were multiple ways to achieve the same result, which helped broaden my perspective and deepen my understanding of the language and tools.

There were also technical difficulties along the way. One specific challenge was handling user account deletion. Since user data was linked to multiple tables through foreign keys, we had to carefully delete related entries from child tables first to avoid constraint errors. Initially, this led to persistent database errors. By discussing the issue with a teammate and reviewing the table relationships together, we were able to identify the correct deletion order and resolve the problem effectively.

Overall, working in a team taught me not only how to collaborate and communicate with others, but also how much I can grow by solving problems together.