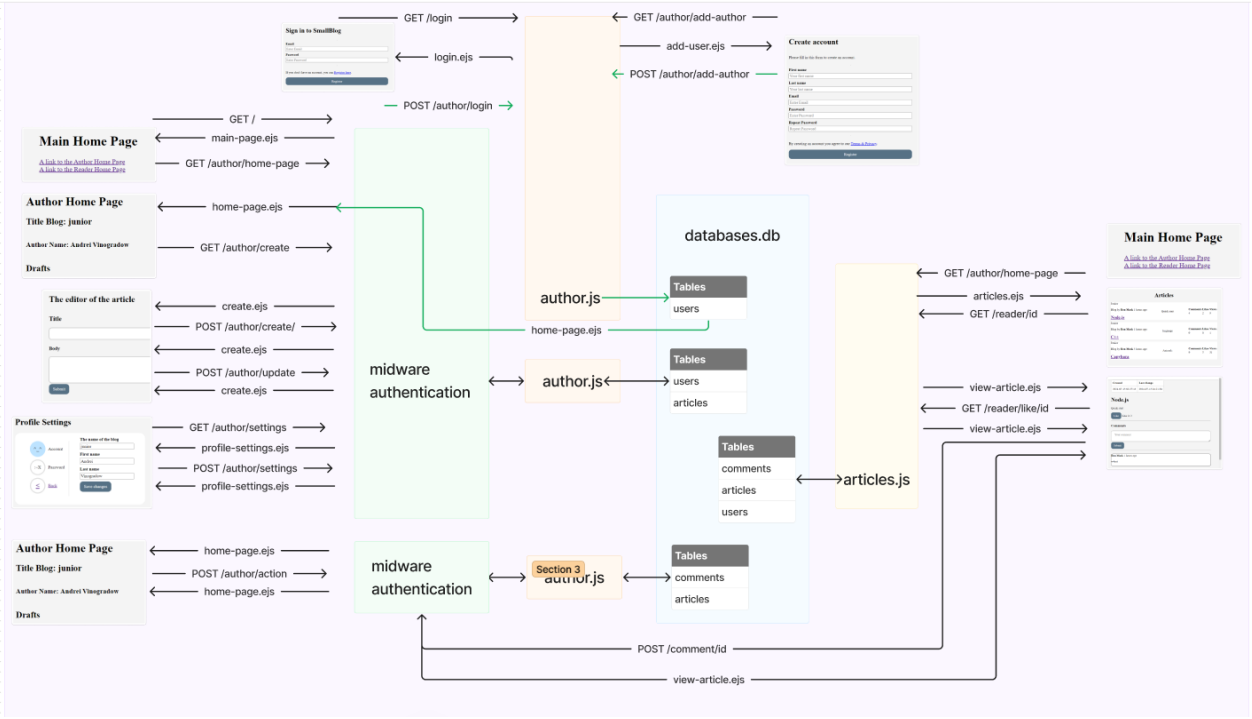
Microblogging website report

This report provides an overview of the microblogging website project, including its architecture, data model, and an implemented extension for user profile management. The website uses Express.js for the server-side, SQLite for data storage, and EJS for templating.

High-Level Schematic Diagram

This high-level schematic diagram illustrates the architecture and flow of a microblogging website, depicting the interactions between the client, server, and database across various pages and functionalities. Here's a detailed description of each component and their interactions:



1. Main home page
   Client Interaction:
   - **GET /main-page**: the main page is rendered using main-page.ejs.
   - the main page provides links to other parts of the application, such as the author home page and reader home page.
2. Author home page
   Client Interaction:
   - **GET /author/home-page**: the author home page is rendered using home-page.ejs.
   - the author can create new articles and update drafts.
3. Middleware authentication
   Purpose: Ensures that users are authenticated before accessing certain routes.
   Client Interaction:
   - the middleware checks authentication for routes such as profile settings and author actions.
4. Profile settings

Client Interaction:

- **GET /author/settings**: renders the profile settings page using profile-settings.ejs
- **POST /author/settings**: submits profile changes, which are then handled by the server to update the user data in the database.

5. Article management
   Client Interaction:
   - **GET /author/create**: renders the article creation page using create.ejs
   - **POST /author/create**: submits new article data to the server.
   - **POST /author/update**: updates existing articles.
6. Login and registration
   Client Interaction:
   - **GET /login**: renders the login page using login.ejs.
   - **POST /author/login**: processes login information
   - **GET /author/add-author**: renders the registration page using add-user.ejs.
   - **POST /author/add-author**: processes new user registration.

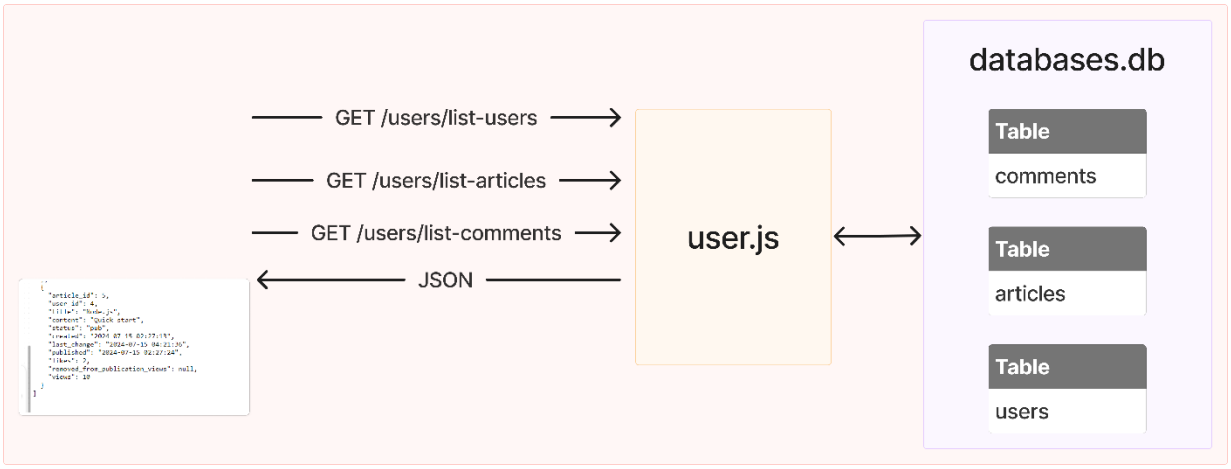
7. Database (SQLite)
   Tables:
   1. **Users Table**: stores user information such as user_id, first_name, last_name, email, and password.
   2. **Articles Table**: stores articles with fields such as article_id, user_id, title, content, and status.
   3. **Comments Table**: stores comments with fields such as comment_id, user_id, article_id, and comment text.

8. View articles and comments
   Client Interaction:
   - **GET /reader/id**: renders a specific article using view-article.ejs.
   - **POST /comment/id**: allows users to post comments on articles, handled by view-article.ejs.
9. article.js and author.js
   Backend Controllers:
   - **author.js**: handles author-related routes and functionalities.
   - **articles.js**: manages article-related routes and functionalities, including retrieving and updating articles.

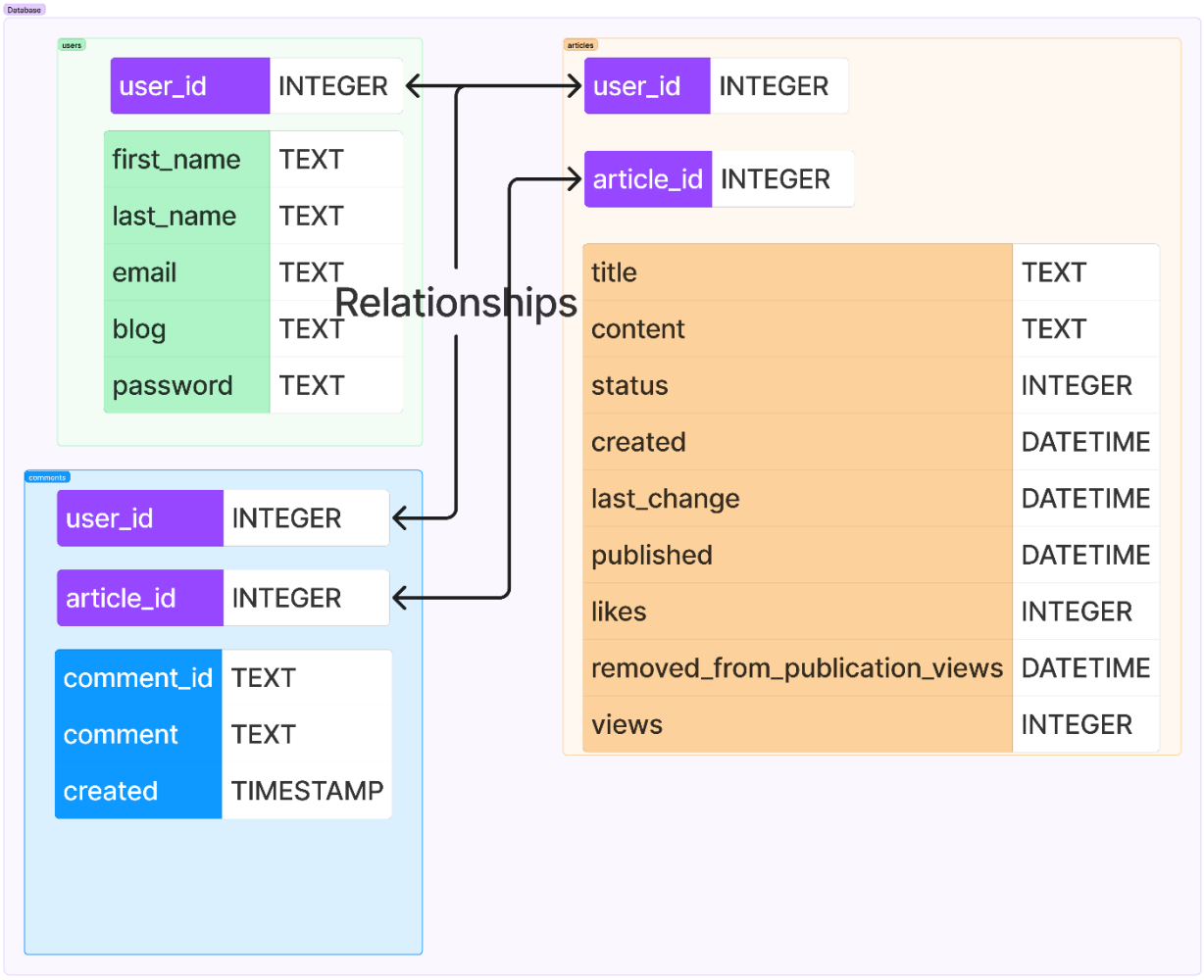
Flow Explanation

1. User authentication:
   Users log in or register through login.ejs and add-user.ejs pages.
   Middleware authentication ensures users are logged in before accessing protected routes.
2. Profile management:
   Users can update their profile information and passwords through the profile settings page.

3. Article creation and management:
   Authors can create and update articles using create.ejs.
   Articles are stored in the database and can be viewed by readers through specific article pages.
4. Comment system:
   Only authorized users can comment on articles.
   Comments are saved in the database and displayed on the article pages.
5. Database interactions:
   User, article, and comment data are managed in the SQLite database.
   Backend controllers (author.js and articles.js) handle CRUD operations for the respective data.



Data Model



The database for this microblogging website consists of three main tables: users, articles, and comments. These tables are interconnected through foreign key relationships to maintain data integrity and establish relationships between the entities.

Relationships

One-to-Many Relationship:

A user can write multiple articles (users to articles). A user can post multiple comments (users to comments). An article can have multiple comments (articles to comments).

The foreign key constraints ensure referential integrity, meaning that each article must be linked to a valid user and each comment must be linked to both a valid user and a valid article. This setup helps maintain the logical consistency of the data throughout the database.

Extension description: user authentication with JWT

Extension implemented.

I implemented user authentication using JSON Web Tokens (JWT). This extension ensures that only authenticated users can access certain routes, thereby protecting sensitive author data from unauthorized access.

Implementation details

1. **JWT token generation**. Upon successful login, a JWT token is generated and sent to the client. This token is used to authenticate subsequent requests.
2. **Middleware for authentication**. Middleware functions are implemented to verify the JWT token on protected routes. This ensures that only authenticated users can access these routes.
3. **Protected routes**. Routes related to creating, updating, and deleting articles, as well as accessing profile settings, are protected. Unauthorized users attempting to access these routes are redirected or receive an error response.

Key Implementation Aspects.

1. JWT token generation and verification:
   The jsonwebtoken library is used to generate and verify tokens.
   Upon login, the server generates a token using a secret key and sends it to the client.
   For protected routes, a middleware function checks the presence and validity of the token.
2. Middleware implementation:
   Middleware functions ensure that the token is present in the request headers and is valid.
   If the token is missing or invalid, the request is denied.
3. Separation of routes:
   Routes are organized such that those requiring authentication are separated and protected by the middleware.
   This structure helps in maintaining clarity and security in route management.

Key highlights

1. Security: the implementation ensures that only authenticated users can access sensitive routes, enhancing the security of the application.
2. Token management: JWT tokens provide a secure and scalable way to manage user sessions and authentication.
3. Middleware utilization: middleware functions streamline the verification process, ensuring that all protected routes are consistently secured without repetitive code.

By integrating JWT authentication and organizing protected routes, the application now maintains a robust security framework that protects user and author data from unauthorized access.