SQLite Database Documentation for Social Media Application

1 Database Used

The application uses **SQLite** as its database, implemented via the `better-sqlite3` Node.js package. SQLite is a lightweight, serverless, file-based relational database that is ideal for small to medium-sized applications due to its simplicity and ease of setup. The database file is named `ourApp.db`, as specified in the line:

```
const db = require("better-sqlite3")("ourApp.db");
```

2 Explanation of Database-Related Code

The database code in the provided application handles the creation, configuration, and interaction with the SQLite database. Below is a detailed breakdown of the database-related code.

2.1 Database Initialization and Configuration

```
const db = require("better-sqlite3")("ourApp.db");
db.pragma("synchronous = FULL");
db.pragma("journal_mode = WAL");
```

- **`better-sqlite3` Initialization**: The `better-sqlite3` package is a synchronous SQLite driver for Node.js, providing a fast and straightforward API for database operations. `db` is the database connection object, and `"ourApp.db"` specifies the database file. If the file doesn't exist, SQLite creates it.
- **PRAGMA Statements**:
  - `synchronous = FULL`: Ensures that all database writes are fully synchronized with the disk, prioritizing data integrity over performance. This setting minimizes the risk of database corruption during crashes but may slow down write operations.
  - `journal_mode = WAL`: Sets the database to use Write-Ahead Logging (WAL) mode. WAL improves concurrency by allowing reads to occur while writes are being committed, reducing contention in multi-user scenarios. It also enhances performance for write-heavy workloads.

2.2 Table Creation

```
const createTables = db.transaction(() => {
    db.prepare(
        `CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL UNIQUE,
        password TEXT NOT NULL,
        isAdmin BOOLEAN DEFAULT 0,
        profilePicture TEXT
        )`
    ).run();
    db.prepare(
        `CREATE TABLE IF NOT EXISTS posts (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        createdDate TEXT,
        title STRING NOT NULL,
        body TEXT NOT NULL,
        image TEXT,
        authorid INTEGER,
        FOREIGN KEY (authorid) REFERENCES users(id)
        )`
    ).run();
    db.prepare(
        `CREATE TABLE IF NOT EXISTS likes (
        userId INTEGER,
        postId INTEGER,
        PRIMARY KEY (userId, postId),
        FOREIGN KEY (userId) REFERENCES users(id),
        FOREIGN KEY (postId) REFERENCES posts(id)
        )`
    ).run();
    db.prepare(
        `CREATE TABLE IF NOT EXISTS comments (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        content TEXT NOT NULL,
        createdDate TEXT,
        userId INTEGER,
        postId INTEGER,
        FOREIGN KEY (userId) REFERENCES users(id),
        FOREIGN KEY (postId) REFERENCES posts(id)
        )`
    ).run();
});
createTables();
```

- **Transaction**: The `db.transaction` method wraps the table creation statements in a single transaction, ensuring that either all tables are created successfully or none are (atomicity). This is useful for maintaining database consistency during initialization.
- **Table Definitions**:

1. **`users` Table**: Stores user information.
   - **Columns**:
     * `id`: Auto-incrementing primary key (unique identifier for each user).
     * `username`: Unique text field for the user's username.
     * `password`: Stores the hashed password (using `bcrypt`).
     * `isAdmin`: Boolean indicating if the user is an admin (defaults to `0` or `false`).
     * `profilePicture`: Text field for storing the path to the user's profile picture (optional).
   - **Constraints**:
     * `username` is `UNIQUE` to prevent duplicate usernames.
     * `NOT NULL` ensures `username` and `password` are required.
2. **`posts` Table**: Stores user posts.
   - **Columns**:
     * `id`: Auto-incrementing primary key.
     * `createdDate`: Text field for storing the post creation date (in ISO format).
     * `title`: Required text field for the post title.
     * `body`: Required text field for the post content.
     * `image`: Optional text field for the path to an uploaded image.
     * `authorid`: Integer referencing the `id` of the user who created the post.
   - **Constraints**: `FOREIGN KEY (authorid) REFERENCES users(id)` ensures that `authorid` corresponds to a valid user in the `users` table.
3. **`likes` Table**: Tracks which users have liked which posts (many-to-many relationship).
   - **Columns**:
     * `userId`: Integer referencing the user who liked the post.
     * `postId`: Integer referencing the liked post.
   - **Constraints**:
     * `PRIMARY KEY (userId, postId)` ensures that a user can like a post only once (composite key).
     * Foreign keys ensure `userId` and `postId` reference valid entries in the `users` and `posts` tables, respectively.
4. **`comments` Table**: Stores comments on posts.

- **Columns**:
  - `id`: Auto-incrementing primary key.
  - `content`: Required text field for the comment text.
  - `createdDate`: Text field for the comment creation date.
  - `userId`: Integer referencing the user who made the comment.
  - `postId`: Integer referencing the post being commented on.
- **Constraints**: Foreign keys ensure `userId` and `postId` reference valid entries in the `users` and `posts` tables.

- `IF NOT EXISTS`: The `IF NOT EXISTS` clause in `CREATE TABLE` prevents errors if the tables already exist, making the code idempotent (safe to run multiple times).

2.3 Database Queries in Routes

The application uses prepared statements with `db.prepare` to execute SQL queries safely, preventing SQL injection. Below are examples of how the database is used in various routes:

- **User Registration (`POST /register`)**:

```
const usernameStatement = db.prepare("SELECT * FROM users
   WHERE username = ?");
const usernameCheck = usernameStatement.get(username);
const hashedPassword = bcrypt.hashSync(password, 10);
const result = db.prepare("INSERT INTO users (username,
   password, isAdmin) VALUES (?, ?, ?)").run(username,
   hashedPassword, 0);
const newUser = db.prepare("SELECT * FROM users WHERE id =
   ?").get(result.lastInsertRowid);
```

  Checks if the username already exists. Inserts a new user with a hashed password and `isAdmin = 0`. Retrieves the newly created user to generate a JWT token.

- **User Login (`POST /login`)**:

```
const user = db.prepare("SELECT * FROM users WHERE username =
    ?").get(username);
```

  Retrieves the user by username to verify credentials.

- **Fetching Posts (`GET /`)**:

```
const postsStatement = db.prepare(`
    SELECT posts.*, users.username, COALESCE(users.
       profilePicture, '/default-profile.png') AS
       profilePicture,
           (SELECT COUNT(*) FROM likes WHERE likes.postId =
              posts.id) AS likeCount,
           EXISTS (SELECT 1 FROM likes WHERE likes.postId =
              posts.id AND likes.userId = ?) AS hasLiked,
```

```
            (SELECT COUNT(*) FROM comments WHERE comments.
               postId = posts.id) AS commentCount
    FROM posts
    INNER JOIN users ON posts.authorid = users.id
    ORDER BY posts.id DESC
`);
posts = postsStatement.all(req.user ? req.user.userid : 0);
```

Retrieves all posts with the author's username, profile picture, like count, whether the current user has liked the post, and comment count. Uses subqueries to compute `likeCount` and `hasLiked`. Joins with the `users` table to get author details.

- **Creating a Post (`POST /create-post`)**:

```
const ourStatement = db.prepare("INSERT INTO posts (title,
   body, image, authorid, createdDate) VALUES (?, ?, ?, ?, ?)
   ");
const result = ourStatement.run(req.body.title, req.body.body
   , imagePath, req.user.userid, new Date().toISOString());
const getPostStatement = db.prepare("SELECT * FROM posts
   WHERE ROWID = ?");
const realPost = getPostStatement.get(result.lastInsertRowid)
   ;
```

Inserts a new post with the provided title, body, image (if uploaded), author ID, and creation date. Retrieves the newly created post to redirect to its page.

- **Liking a Post (`POST /like/:id`)**:

```
const insertStatement = db.prepare("INSERT OR IGNORE INTO
   likes (userId, postId) VALUES (?, ?)");
insertStatement.run(userId, postId);
```

Inserts a like entry, with `OR IGNORE` preventing duplicate likes due to the composite primary key.

- **Deleting a Post (`POST /delete-post/:id`)**:

```
const deleteLikes = db.prepare("DELETE FROM likes WHERE
   postId = ?");
deleteLikes.run(req.params.id);
const deleteComments = db.prepare("DELETE FROM comments WHERE
    postId = ?");
deleteComments.run(req.params.id);
const deleteStatement = db.prepare("DELETE FROM posts WHERE
   id = ?");
deleteStatement.run(req.params.id);
```

Deletes associated likes and comments before deleting the post to maintain referential integrity.

- **Admin Deleting a User (`POST /admin/delete-user/:id`)**:

```
const deletePosts = db.prepare("DELETE FROM posts WHERE
   authorid = ?");
deletePosts.run(userId);
const deleteLikes = db.prepare("DELETE FROM likes WHERE
   userId = ?");
deleteLikes.run(userId);
const deleteComments = db.prepare("DELETE FROM comments WHERE
    userId = ?");
deleteComments.run(userId);
const deleteUser = db.prepare("DELETE FROM users WHERE id =
   ?");
deleteUser.run(userId);
```

Cascades deletion of all user-related data (posts, likes, comments) before deleting the user.

2.4 Key Features of Database Usage

- **Prepared Statements**: The use of `db.prepare` with parameterized queries (`?`) prevents SQL injection by safely handling user input. Example: `db.prepare("SELECT * FROM users WHERE username = ?").get(username)`.
- **Foreign Keys**: Foreign key constraints ensure referential integrity (e.g., a post's `authorid` must reference a valid user). SQLite requires `PRAGMA foreign_keys = ON` to enforce foreign keys, which is not explicitly set in the code but is assumed to be enabled by default in `better-sqlite3`.
- **Data Sanitization**: User inputs (e.g., post titles, bodies, comments) are sanitized using `sanitizeHTML` to prevent XSS attacks before being stored in the database.
- **Efficient Queries**: Subqueries and joins are used to fetch related data (e.g., like counts, user details) in a single query, reducing database round-trips. Example: The `posts` query in the dashboard route fetches post details, author info, and like/comment counts in one go.

2.5 Potential Improvements

- **Foreign Key Enforcement**: Explicitly enable foreign key support with `db.pragma("foreign_ke = ON")` to ensure constraints are enforced.
- **Error Handling**: Add try-catch blocks around database operations to handle potential errors (e.g., database corruption, constraint violations).
- **Indexing**: Create indexes on frequently queried columns (e.g., `posts.authorid`, `likes.postId`) to improve query performance for large datasets.
- **Connection Management**: Ensure the database connection is properly closed when the application shuts down (e.g., using `db.close()`).
- **Schema Migrations**: Use a migration tool (e.g., `knex.js`) to manage schema changes over time, especially if the application evolves.

3 Summary

The application uses **SQLite** with the `better-sqlite3` package to manage a simple social media platform's data. The database schema includes tables for `users`, `posts`, `likes`, and `comments`, with appropriate primary and foreign key constraints. The code employs prepared statements, transactions, and sanitization for security and efficiency. The database is configured for data integrity (`synchronous = FULL`) and concurrency (`journal_mode = WAL`), making it suitable for a small-scale application.