Backend: blog-service

Implement the likeController

File to be edited: likeController.js in controllers folder

This guide outlines the steps to implement the `Like` controller functions: `addLike`, `removeLike`, and `getLikesByPost`. These functions handle adding likes, removing likes, and fetching likes for a specific post.

---

1. Setup and Imports

- **Objective**: Import necessary modules and models.
- **Steps**:
    1. Import the `Like` model to interact with the likes collection:
    2. Import the `Post` model to interact with the posts collection:

---

2. Implement the `addLike` Function

- **Function Name**: `addLike`
- **Objective**: Add a like to a specific post by the current user.
- **Steps**:
    1. Use `Post.findById(req.params.id)` to check if the post exists.
        - Return a `404 Not Found` error if the post does not exist.
    2. Use `Like.findOne()` to check if the user has already liked the post.
        - Return a `400 Bad Request` error if a like already exists for this user and post.
    3. Create a new `Like` document with the `user` (from `req.user.id`) and `post` (from `req.params.id`).
    4. Save the `Like` document using `like.save()`.
    5. Update the post:
        - Add the `like._id` to the `post.likes` array.
        - Save the updated post using `post.save()`.
    6. Respond with a `201 Created` status and the created like.

---

3. Implement the `removeLike` Function

- **Function Name**: `removeLike`
- **Objective**: Remove a like from a specific post by the current user.
- **Steps**:
    1. Use `Like.findOne()` to find the like document for the current user and post.
        - Return a `404 Not Found` error if the like does not exist.
    2. Check if the `like.user` matches `req.user.id`.
        - Return a `403 Forbidden` error if the current user is not authorized to remove the like.
    3. Delete the like using `like.deleteOne()`.
    4. Update the post:
        - Remove the `like._id` from the `post.likes` array using the `filter` method.
        - Save the updated post using `post.save()`.
    5. Respond with a success message.

---

4. Implement the `getLikesByPost` Function

- **Function Name**: `getLikesByPost`
- **Objective**: Fetch all likes for a specific post.
- **Steps**:
    1. Use `Like.find()` to retrieve all likes where the `post` matches `req.params.id`.
    2. Use `populate()` to include the user's `name` and `email` fields in the result.
    3. Respond with the retrieved likes in JSON format.

---

5. Error Handling

- **Objective**: Ensure robust error handling for each function.
- **Steps**:
    1. Wrap the logic of each function in a `try...catch` block.
    2. Log any errors to the console for debugging.
    3. Respond with a `500 Internal Server Error` status and a descriptive error message in case of failure.

Implement the likes Router

File to be edited: likeRoutes.js in routes folder

This guide outlines the steps to implement the provided routes for managing likes on posts. These routes handle adding likes, removing likes, and fetching likes for a specific post.

---

1. Set Up Dependencies

- **Objective**: Ensure necessary modules and middleware are available.
- **Steps**:
    1. Import the `express` module to create a router.
    2. Import the `addLike`, `removeLike`, and `getLikesByPost` functions from the `likeController`.
    3. Import the `protect` middleware to ensure only authenticated users can interact with the routes.
    4. Import the `logger` module for logging requests.

---

2. Create a Router

- **Objective**: Use `express.Router()` to define and manage routes for likes.
- **Steps**:
    1. Initialize a router instance using `express.Router()`.

---

3. Add Logging Middleware

- **Objective**: Log incoming requests for debugging and monitoring.
- **Steps**:
    1. Use `router.use()` to add a middleware function that logs the HTTP method and URL of each request.
    2. Use the `logger.info` method to log the message in a structured format.
    3. Call `next()` to pass control to the next middleware or route handler.

---

4. Define the Routes

- **Objective**: Implement routes for adding, removing, and fetching likes.

- **Steps**:
    1. **Add Like**:
        - Use `router.post()` to define a route for adding a like to a post.
        - Apply the `protect` middleware to ensure only authenticated users can add likes.
        - Attach the `addLike` function as the route handler.
    2. **Remove Like**:
        - Use `router.delete()` to define a route for removing a like from a post.
        - Apply the `protect` middleware to ensure only authenticated users can remove likes.
        - Attach the `removeLike` function as the route handler.
    3. **Fetch Likes**:
        - Use `router.get()` to define a route for fetching likes for a specific post.
        - Apply the `protect` middleware to ensure only authenticated users can view likes.
        - Attach the `getLikesByPost` function as the route handler.

---

5. Secure the Routes

- **Objective**: Ensure only authorized users can access these routes.
- **Steps**:
    1. Use the `protect` middleware to validate the user's authentication token.
    2. Ensure the middleware attaches user information (e.g., `req.user`) to the request object for further use in controllers.

---

6. Export the Router

- **Objective**: Make the router available for integration with the main application.
- **Steps**:
    1. Use `module.exports` to export the router instance.
    2. Import and mount this router in your main application file (e.g., `app.js`) under the desired base path.

Note: Update routes in server.js in blog-service according to the requirement.

API gateway: Implementation for `/likes` Route

File to be edited: gatewayRoutes.js

This guide outlines how to implement the `/likes` route function, which forwards requests to the blog service's likes API. It ensures secure interaction with the backend, handles different HTTP methods, and provides proper error handling.

---

1. Set Up Middleware for the Route

- **Objective**: Ensure the route is protected by middleware.
- **Steps**:
    1. Use the `protect` middleware to secure the `/likes` route.
        - This middleware should validate the user's authentication token and attach user data to the `req` object.
    2. Ensure the middleware is applied before the route handler.

---

2. Construct the Target URL

- **Objective**: Dynamically create the URL to forward requests to the likes API in the blog service.
- **Steps**:
    1. Extract the base URL for the blog service from the environment variable (`BLOG_SERVICE_URL`).
    2. Append `/api/likes` to the base URL.
    3. Add the dynamic part of the request URL (`req.url`) to complete the full target URL.

---

3. Forward the Request Using Axios

- **Objective**: Relay the incoming request to the blog service.
- **Steps**:
    1. Use `axios` to send the request to the target URL.
    2. Forward the following from the incoming request:
        - **Method**: Use `req.method` to preserve the original HTTP method (e.g., GET, POST, DELETE).

    - **Headers**: Include the `Authorization` header from `req.headers.authorization` for user authentication.
    - **Body**: Forward `req.body` for requests like POST or PUT.
3. Log the forwarded request using `logger.info` for traceability.

---

4. Handle the Response

- **Objective**: Relay the response from the blog service back to the client.
- **Steps**:
    1. Check if the blog service returns a successful response.
    2. Send the status and data from the blog service response to the client using `res.status(response.status).json(response.data)`.

---

5. Handle Errors

- **Objective**: Provide robust error handling for different failure scenarios.
- **Steps**:
    1. Use a `try...catch` block to catch errors during the request forwarding process.
    2. Log the error using `logger.error` with details about the HTTP method, target URL, and error message.
    3. Use optional chaining (`error.response?.status`) to determine the status code for the error response:
        - If the blog service responds with an error, forward the same status code and message to the client.
        - If no specific status code is available, default to a `500 Internal Server Error`.
    4. Include a user-friendly error message in the JSON response to the client.

Frontend: Like Functionality

File to be edited: LikeButton.jsx

Implement `handleLikeClick` and `fetchLikes`

This guide outlines how to implement the `fetchLikes` and `handleLikeClick` functions. Follow the steps carefully to ensure proper functionality and error handling.

---

1. Implement the `fetchLikes` Function

- **Objective**: Retrieve the current like count and the user's like status for the specified post.
- **Steps**:
    1. **Update the Function**:
        - Update the function `fetchLikes` within the `useEffect` hook.
    2. **Access Authentication Token**:
        - Retrieve the `auth_user` object from `localStorage` and parse it.
        - Extract the `token`. If the `token` is missing, alert the user and stop execution.
    3. **Fetch Likes from the API**:
        - Use the `fetch` API to send a `GET` request to the endpoint: `${import.meta.env.VITE_API_URL}/api/likes/${postId}`.
        - Include the `Authorization` header with the token for user authentication.
    4. **Handle API Response**:
        - Check if the response is successful (`response.ok`).
        - Parse the response as JSON to extract the likes data.
    5. **Update State**:
        - Use the `length` of the likes data array to update the `likes` state.
        - Determine if the current user has liked the post by checking if their user ID exists in the likes data. Update the `isLiked` state accordingly.
    6. **Handle Errors**:
        - Use a `try...catch` block to catch and log any errors. Alert the user in case of an error.

---

2. Implement the `handleLikeClick` Function

- **Objective**: Allow the user to like or unlike a post and update the UI accordingly.
- **Steps**:
    1. **Update the Function**:
        - Update the function `handleLikeClick`.
    2. **Access Authentication Token**:
        - Retrieve the `auth_user` object from `localStorage` and parse it.
        - Extract the `token`. If the `token` is missing, alert the user and stop execution.
    3. **Determine HTTP Method**:

    - Use `POST` if the user has not liked the post yet.
    - Use `DELETE` if the user is undoing a like.
4. **Construct API URL**:
    - Use the endpoint: `${import.meta.env.VITE_API_URL}/api/likes/${postId}`.
5. **Send API Request**:
    - Use the `fetch` API to send the request.
    - Include the `Authorization` header with the token for user authentication.
    - Check if the response is successful. If not, parse the error response and throw an error with a meaningful message.
6. **Update Like Count**:
    - Fetch the updated like count by calling the same API endpoint as in `fetchLikes`.
    - Update the `likes` and `isLiked` states based on the response.
7. **Handle Errors**:
    - Use a `try...catch` block to log and alert the user in case of an error.