Backend: blog-service

Implement the CommentsController

File to be edited: commentsController.js in controllers folder

This guide explains how to implement the `commentsController` logic, including fetching, adding, editing, and deleting comments for a specific post.

---

1. Import Models

- **Objective**: Use Mongoose models for comments and posts.
- **Steps**:
    1. Ensure `Comment` and `Post` models are imported.

---

2. Fetch All Comments for a Post

- **Objective**: Retrieve all comments associated with a specific post ID.
- **Steps**:
    1. Use `Comment.find()` to fetch comments where `post` matches `req.params.id`.
    2. Use `populate()` to include details about the `author` (e.g., name and email).
    3. Handle errors by wrapping the logic in a `try...catch` block.
    4. Return the fetched comments as a JSON response.

---

3. Fetch a Single Comment by ID

- **Objective**: Retrieve a specific comment using its ID.
- **Steps**:
    1. Use `Comment.findById()` to fetch the comment by `req.params.id`.
    2. Use `populate()` to include author details.
    3. Check if the comment exists; if not, return a `404 Not Found` response.
    4. Handle errors and return a JSON response with the comment or error message.

---

4. Add a New Comment

- **Objective**: Add a comment to a specific post.
- **Steps**:
    1. Extract `content` from `req.body`.
    2. Use `Post.findById()` to verify the existence of the post associated with `req.params.id`.
    3. Create a new comment with the `content`, `author` (from `req.user.id`), and `post` ID.
    4. Save the comment using `comment.save()`.
    5. Update the post's `comments` array by pushing the new comment's ID and saving the post.
    6. Handle errors and return a success response with the created comment.

---

5. Edit a Comment

- **Objective**: Allow the author to edit their comment.
- **Steps**:
    1. Use `Comment.findById()` to fetch the comment by `req.params.id`.
    2. Verify the comment exists; if not, return a `404 Not Found` response.
    3. Check if the logged-in user (`req.user.id`) matches the comment's author.
    4. Update the comment's `content` with the new value from `req.body.content`.
    5. Save the updated comment using `comment.save()`.
    6. Handle errors and return a success response with the updated comment.

---

6. Delete a Comment

- **Objective**: Allow the author to delete their comment.
- **Steps**:
    1. Use `Comment.findById()` to fetch the comment by `req.params.id`.
    2. Verify the comment exists; if not, return a `404 Not Found` response.
    3. Check if the logged-in user (`req.user.id`) matches the comment's author.
    4. Use `comment.deleteOne()` to delete the comment from the database.
    5. Update the associated post by removing the comment ID from its `comments` array using `Post.updateOne()` with `$pull`.
    6. Handle errors and return a success response.

---

7. Integrate with Routes

- **Objective**: Connect these controller functions to the Express routes.
- **Steps**:
    1. Import these functions into the comments router file.
    2. Define the routes and attach the corresponding controller functions:
        - `GET /comments/:id` → `getComments`
        - `POST /comments/:id` → `addComment`
        - `PUT /comments/:id` → `editComment`
        - `DELETE /comments/:id` → `deleteComment`

Implement the Comments Router

File to be edited: commentRoutes.js in routes folder

This guide outlines the steps needed to implement the `commentRouter` functionality. The router handles routes for fetching, adding, editing, and deleting comments, and integrates middleware for authentication and logging.

---

1. Set Up the Router

- **Objective**: Create an Express router to handle comment-related operations.
- **Steps**:
    1. Import the `express` module.
    2. Use `express.Router()` to create a new router instance.

---

2. Import Required Modules

- **Objective**: Ensure necessary dependencies are available.
- **Steps**:
    1. Import the controller functions (`getComments`, `addComment`, `editComment`, `deleteComment`) from the `commentsController` file.
    2. Import the `protect` middleware for authentication.
    3. Import a logger instance for request logging.

---

3. Add Middleware for Logging

- **Objective**: Log incoming requests for better traceability.
- **Steps**:
    1. Use `router.use()` to define a middleware that logs the request method and URL.
    2. Call `logger.info()` with a formatted log message (e.g., `GET /api/posts/:id/comments - Request received`).
    3. Call `next()` to pass control to the next middleware or route handler.

---

4. Define Routes

- **Objective**: Handle CRUD operations for comments.
- **Steps**:
    1. **Fetch Comments (`GET /:id`)**:
        - Use `router.get()` to define a route for fetching comments by post ID.
        - Apply the `protect` middleware to ensure only authenticated users can access this route.
        - Pass the `getComments` controller function as the route handler.
    2. **Add a Comment (`POST /:id`)**:
        - Use `router.post()` to define a route for adding a comment to a post.
        - Apply the `protect` middleware.
        - Pass the `addComment` controller function as the route handler.
    3. **Edit a Comment (`PUT /:id`)**:
        - Use `router.put()` to define a route for editing an existing comment by its ID.
        - Apply the `protect` middleware.
        - Pass the `editComment` controller function as the route handler.
    4. **Delete a Comment (`DELETE /:id`)**:
        - Use `router.delete()` to define a route for deleting a comment by its ID.
        - Apply the `protect` middleware.
        - Pass the `deleteComment` controller function as the route handler.

---

5. Export the Router

- **Objective**: Make the router available to the application.
- **Steps**:
    1. Use `module.exports` to export the router instance.

2. Ensure the router is imported and mounted in the main application file (e.g., `app.js`).

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

API gateway: Implementation for `/comments` Route

File to be edited: gatewayRoutes.js

This guide will help students implement the `/comments` route in an Express.js application. The route will forward requests related to comments to the blog service, ensuring authentication and proper error handling.

---

1. Set Up the Route

- Create a new route under the same router or a separate router file if necessary.
- Use `router.use()` to handle all HTTP methods (GET, POST, PUT, DELETE) for `/comments`.
- Import necessary modules such as `express`, `axios`, and any middleware (e.g., `protect`).

---

2. Apply Authentication Middleware

- Ensure the route is protected by using the `protect` middleware.
- This middleware should verify that the user is authenticated before processing the request further.

---

3. Construct the Blog Service URL

- Use the environment variable `BLOG_SERVICE_URL` to form the target URL.
- Append `/api/comments` and the dynamic parts of the incoming request URL (`req.url`) to this base URL.
- Example: If `BLOG_SERVICE_URL` is `http://localhost:5000` and the request URL is `/comments/123`, the full URL will be `http://localhost:5000/api/comments/123`.

---

4. Forward the Request

- Use `axios` to forward the incoming request to the constructed URL.
- Pass the following properties from the original request:
  - **HTTP Method**: Use `req.method` to maintain the same HTTP verb (e.g., GET, POST).
  - **Headers**: Include the `Authorization` header from the original request.
  - **Body**: Pass `req.body` for methods like POST and PUT.

---

5. Handle the Response

- On a successful response from the blog service:
  1. Log a message indicating the successful forwarding of the request.
  2. Return the status and data from the blog service back to the client using `res.status(response.status).json(response.data)`.

---

6. Handle Errors

- Use a `try...catch` block to handle errors.
- In the `catch` block:
  1. Log the error using `logger.error` for debugging.
  2. Use optional chaining (`error.response?.status`) to safely access the error status.
  3. Send an appropriate error message back to the client with a status code:
     - Use the status code from the blog service if available.
     - Default to a `500` (Internal Server Error) if no specific status is provided.

Frontend: Comment Functionality

File to be edited: CommentSection.jsx

This guide will help to implement the **fetching**, **adding**, **updating**, and **deleting** functionalities for the `CommentSection` component. Follow each step carefully, testing as you go to ensure correct functionality.

---

1. Fetching Comments

- **Objective**: Display comments for a specific post when the component loads or when the `postId` changes.
- **Steps**:
    1. Edit **fetchcomments** function to fetch comments.
    2. Access the authentication token from `localStorage`. If the token does not exist, alert the user and halt further execution.
    3. Use the `postId` to construct the API URL. This URL should point to the endpoint responsible for fetching comments.
    4. Make a GET request to the API with the token included in the `Authorization` header.
    5. Parse the response data and update the component's `comments` state with the fetched data.
    6. Handle any errors during the request gracefully, logging them for debugging purposes.

---

2. Adding a New Comment

- **Objective**: Allow users to submit new comments.
- **Steps**:
    1. Update **handleAddComment** function.
    2. Prevent the default form submission behavior.
    3. Access the authentication token and the user ID from `localStorage`. If the token is missing, notify the user to log in.
    4. Construct the API endpoint using the `postId`.
    5. Use a POST request to send the new comment content to the server. Ensure the content is properly formatted in JSON.
    6. If the API call is successful, add the new comment to the `comments` state.
    7. Clear the input field after successfully adding the comment.
    8. Handle any errors gracefully, displaying user-friendly error messages.

---

3. Updating an Existing Comment

- **Objective**: Allow users to modify an existing comment.
- **Steps**:
    1. Update **handleUpdateComment** function to handle updates.

2. Prevent the default form submission behavior.
3. Retrieve the authentication token from `localStorage`. If the token is missing, alert the user.
4. Use the `editComment` state to get the comment's ID and construct the API URL for the specific comment.
5. Send a PUT request to the API with the updated comment content in the body.
6. If successful, update the corresponding comment in the `comments` state to reflect the changes.
7. Clear the edit mode by resetting the `editComment` and `newComment` states.
8. Ensure error handling is robust, with clear messages for the user in case of failure.

---

4. Deleting a Comment

- **Objective**: Enable authorized users to delete comments.
- **Steps**:
  1. Update **handleDeleteCommen**t function.
  2. Access the authentication token and current user ID from `localStorage`.
  3. Ensure that the user is authorized to delete the comment (e.g., they are the author or the post owner).
  4. Use the comment's ID to construct the API URL.
  5. Send a DELETE request to the API with the token in the `Authorization` header.
  6. If successful, remove the deleted comment from the `comments` state to update the UI.
  7. Handle any errors, providing meaningful feedback to the user.