Week 8 Day 5 - REST APIs for Blog Application

Create a Node.js application using Express to manage a blogging platform. The API will provide functionality for managing blog posts, comments, likes, and user interactions. For simplicity, all data will be stored in an in-memory array. This assignment focuses on backend logic, with the option to integrate a database in a more advanced version.

Requirements

Project Setup

1. Create a new directory for your project.
2. Navigate to the directory in your terminal.
3. Initialize a Node.js project using `npm init -y`.
4. Install Express: `npm install express`.
5. Create a file named `app.js` for the main application.

API Design

1. Blog Content Management

Data Storage:

- Use an in-memory array to store posts. Each post object should include:
- `id`: Unique identifier (e.g., use a counter or UUID library).
- `title`: Blog title (string).
- `author`: Author name (string).
- `publicationDate`: Date of publication (use the current date for new posts).
- `readTime`: Estimated reading time in minutes (calculated as `Math.ceil(content.length / 200)` assuming 200 words/min).
- `content`: Content of the blog post (string).
- `likes`: Number of likes (default: 0).
- `comments`: Array of comment objects.
- Comments should be stored in an array within each post object. Each comment should have:
- `id`: Unique identifier.
- `author`: Author of the comment.
- `content`: Content of the comment.

Endpoints:

- `GET /posts`

Retrieves all blog posts.
Implementation:
- Fetch the array of posts and send it as a JSON response using `res.json(posts)`.
- Include all post properties, including the nested comments array.
- `GET /posts/:id`

Retrieves a specific blog post by its ID.
Implementation:
- Parse the `id` from the request parameters.
- Find the post in the array using `Array.find()`.
- If found, return the post as a JSON response.
- If not found, respond with a `404` status and a message: `{ error: "Post not found" }`.
- `POST /posts`

Creates a new blog post.
Implementation:
- Parse the `title`, `author`, and `content` from the request body.
- Validate that all required fields are provided. Respond with `400 Bad Request` if any are missing.
- Generate the following fields:
- `id`: Use a counter or UUID.
- `publicationDate`: Set to the current date.
- `readTime`: Calculate based on content length.
- `likes`: Initialize to 0.
- `comments`: Initialize to an empty array.
- Push the new post object into the in-memory array.
- Respond with the newly created post and status `201`.
- `PUT /posts/:id`

Updates an existing blog post by ID.
Implementation:
- Parse the `id` from the request parameters.
- Find the post using `Array.find()`.
- If not found, respond with a `404` status.
- Update the fields provided in the request body (`title`, `content`, etc.).
- Respond with the updated post as a JSON response.

- `DELETE /posts/:id`
Deletes a specific blog post by ID.
Implementation:
- Parse the `id` from the request parameters.
- Find the post using `Array.findIndex()`.
- If not found, respond with a `404` status.
- Remove the post using `Array.splice()`.
- Respond with a success message: `{ message: "Post deleted successfully" }`.

2. User Interactions

Endpoints:

- `POST /posts/:id/like`
Toggles a like for a specific blog post.
Implementation:
- Parse the `id` from the request parameters.
- Find the post using `Array.find()`.
- If not found, respond with a `404` status.
- Increment or decrement the `likes` count as needed.
- Respond with the updated like count: `{ likes: <count> }`.
- `POST /posts/:id/comment`
Adds a comment to a specific blog post.
Implementation:
- Parse the `id` from the request parameters.
- Parse `author` and `content` from the request body.
- Validate the input fields. Respond with `400 Bad Request` if any are missing.
- Generate a unique `id` for the comment.
- Append the comment to the post's `comments` array.
- Respond with the updated post or a `404` if the post is not found.
- `GET /posts/:id/comments`
Retrieves all comments for a specific blog post.
Implementation:
- Parse the `id` from the request parameters.
- Find the post using `Array.find()`.
- If not found, respond with a `404` status.
- Respond with the `comments` array.

- `PUT /comments/:commentId`

  Updates a specific comment by ID.
  Implementation:
- Parse the `commentId` from the request parameters.
- Iterate over all posts to find the comment.
- If the comment is found, update the `content` field.
- Respond with the updated comment or a `404` if not found.
- `DELETE /comments/:commentId`

  Deletes a specific comment by ID.
  Implementation:
- Parse the `commentId` from the request parameters.
- Iterate over all posts to find and remove the comment.
- If not found, respond with a `404` status.
- Respond with a success message.

3. Search and Filter

Endpoints:

- `GET /search`

  Searches blog posts by title or content.
  Implementation:
- Parse the query parameter (`q`) from the request.
- Filter posts using `Array.filter()` for matches in `title` or `content`.
- Respond with matching posts or an empty array.
- `GET /filter`

  Filters blog posts by tags or author.
  Implementation:
- Parse query parameters (e.g., `?author=John&tag=tech`).
- Filter posts based on the criteria.
- Respond with matching posts or an empty array.

Middleware Implementation

1. Request Logger
   Logs the method and URL of each incoming request.
2. JSON Body Parser
   Use `express.json()` to parse JSON payloads.
3. Input Validation
   Validate required fields for POST and PUT requests.

4. 404 Handler
   Handles invalid routes with a JSON response: `{ error: "Resource not found" }`.
5. Error Handling Middleware
   Logs errors and sends a generic "Internal Server Error" message for unhandled errors.

Testing

Use Postman or curl to test all API endpoints. Ensure you:

1. Create, update, and delete blog posts.
2. Add and manage comments for posts.
3. Like/unlike blog posts and validate counts.
4. Search and filter blog posts.
5. Test error handling for invalid requests.