

BLOG POST APPLICATION - STATIC ANALYSIS REPORT

Software Quality Engineering

(SE-G)



Submitted by

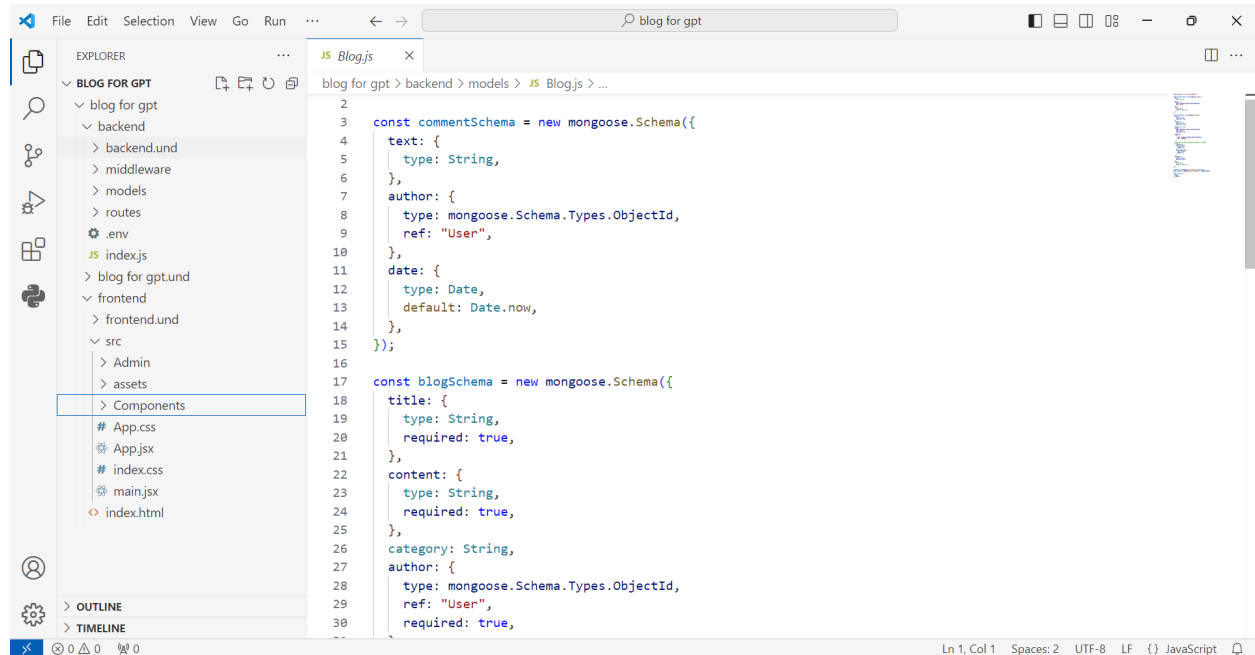
Amna Shahid (21I-1148)
Fatima Qurban (21I-1195)
Ushna Nadeem (21I-1225)

Submitted to

Ma'am Saba Kanwal

Report

Directory Structure



The project(Blog Post Application) has a well-structured layout. Here's a quick breakdown:

Backend

Each directory has different files:

- **Middleware:** It contains authmiddleware.js.
- **Models:** It contains Blog.js and User.js.
- **Routes:** This directory includes following files
 - adminRoutes.js
 - blogRoutes.js
 - searchRoutes.js
 - userRoutes.js
- **Main Backend Entry Point:** index.js.

Frontend

- **Main Files:** App.jsx, main.jsx, App.css, and index.css.
- **Components Directory:** It includes React components in further directories.
- **Assets Directory:** For media, icons, or images.

Static Analysis

authmiddleware.js



```
1 const jwt = require("jsonwebtoken");
2 // cookie
3 const cookie = require("cookie");
4 // dotenv
5 require("dotenv").config();
6
7
8
9 function authMiddleware(req, res, next) {
10   // Get token from request header
11   const token = req.cookies.jwt;
12
13   // Check if token exists
14   if (!token) {
15     return res.status(401).json({ msg: "No token, authorization denied" });
16   }
17
18   try {
19     // Verify token
20     const decoded = jwt.verify(token, process.env.JWT_SECRET);
21     req.user = decoded.user;
22     next();
23   } catch (err) {
24     console.log(err);
25     res.status(401).json({ msg: "Token is not valid" });
26   }
27 }
28
29 module.exports = authMiddleware;
```

The `authmiddleware.js` file implements JWT-based authentication middleware.

Code Summary

- **Purpose:** Acts as a Middleware to authenticate API requests.
- **Operations Performed:**
 1. **Token Extraction:** Retrieves the JWT from `req.cookies.jwt`.
 2. **Validation:** Checks if the token exists.
 3. **Verification:** Uses `jsonwebtoken` to decode the token with `JWT_SECRET`.
 4. **Error Handling:** Returns a “401” Unauthorized error if the token is missing or invalid.

General Observations

Following are the strengths and issues of the project:

1. **Strengths:**
 - Proper usage of environment variables (`process.env.JWT_SECRET`) for secret key management.

- Handles missing tokens gracefully by returning an appropriate error message.
- Passes the decoded user information (req.user) to the next middleware.

2. Potential Issues:

- **Error Specificity:** The 401 Unauthorized response is very generic and could be enhanced to differentiate between missing tokens and invalid tokens.
- **No Token Expiry Check:** The middleware does not check if the token has expired, which is critical for security.
- **No Revocation Logic:** There's no handling for token revocation (e.g., logout scenario).

Schema Structure

Blog.js

```

1  const mongoose = require("mongoose");
2
3  const commentSchema = new mongoose.Schema({
4    text: {
5      type: String,
6    },
7    author: {
8      type: mongoose.Schema.Types.ObjectId,
9      ref: "User",
10   },
11   date: {
12     type: Date,
13     default: Date.now,
14   },
15 });
16
17 const blogSchema = new mongoose.Schema({
18   title: {
19     type: String,
20     required: true,
21   },
22   content: {
23     type: String,
24     required: true,
25   },
26   category: String,
27   author: {
28     type: mongoose.Schema.Types.ObjectId,
29     ref: "User",
30   },
31 });

```

The **Blog.js** file defines the schema for blog posts using Mongoose.

Code Summary

- **Purpose:** Defines the structure and relationships for blog posts in the MongoDB database.

- **Key Features:**

- **Schema for Comments:** A nested schema for comments, including text, author, and date.
- **Blog Schema:** Contains fields like title, content, category, author, and comments.
- **Relationships:**
 - References the User model for authors and comment authors.
- **Default Values:** Sets a default Date.now for comment dates.

General Observations

Following are the strengths and issues of the project:

1. Strengths:

- The use of "User" ensures relational integrity by linking blogs and comments to users.
- The schema is well-structured and includes necessary fields for basic blog functionality.

2. Potential Issues:

- **Validation:**
 - Basic validation is present (e.g., "required: true" for title and content), but there's no further validation for length or acceptable characters.
- **Security Concerns:**
 - content and title fields may be susceptible to XSS attacks if user input is not sanitized during insertion or retrieval.
- **Comment Schema Flexibility:**
 - Comments do not enforce a required text field, which could result in empty or incomplete records.

Recommendations

1. Add field-level validation for title and content (e.g., min/max length, acceptable characters).
2. Implement middleware or a plugin to sanitize user inputs before saving to the database.

3. Ensure the comments array includes validation to prevent incomplete comment records.

blogRoutes.js

```
1 const express = require("express");
2 const router = express.Router();
3 const Blog = require("../models/Blog").Blog;
4 const Comment = require("../models/Blog").Comment;
5 const User = require("../models/User");
6 const mongoose = require("mongoose");
7
8 // !Create a new blog post
9 router.post("/", (req, res) => {
10   const blogPost = new Blog({
11     title: req.body.title,
12     content: req.body.content,
13     author: req.body.author,
14   });
15
16   blogPost
17     .save()
18     .then((data) => {
19       res.json(data);
20     })
21     .catch((err) => {
22       res.status(400).json({ message: err.message });
23     });
24 });
25
26 // !GET all blog posts
27 router.get("/", async (req, res) => {
28   //if blog is disabled then don't show it
29   try {
```

The **blogRoutes.js** file handles routes related to blog operations

Code Summary

- **Purpose:** Defines routes for creating, retrieving, and managing blog posts.
- **Key Features:**
 - **Create Route (POST /):** Allows users to create a new blog post.
 - **Database Operations:**
 - Saves a new blog post to the database.
 - Uses the Blog model to handle CRUD operations.
 - **Request Handling:**
 - Extracts data from req.body for blog creation.

Manual Observations

1. **Strengths:**
 - Uses the Blog model to interact with the database, adhering to a structured MVC approach.

- Sends a JSON response containing the saved blog post, which is useful for API consumers.

2. Potential Issues:

- **Input Validation:**
 - There is no validation or sanitization for “req.body” fields like title, content, and author, making the route vulnerable to:
 - **SQL Injection-like Attacks** (although using MongoDB).
- **Error Handling:**
 - The catch block for save() is missing or incomplete. Any database errors would result in an unhandled rejection.
- **Database Overhead:**
 - Large inputs for content are not restricted, potentially leading to storage or performance issues.

Recommendations

1. Add middleware to validate and sanitize req.body fields.
 - Example: Use libraries like express-validator etc.
2. Implement comprehensive error handling:
 - Include a catch block for database save failures.
 - Return appropriate HTTP status codes (e.g., 400 for bad requests, 500 for server errors).
3. Restrict content size to a reasonable length to prevent storage abuse.