CODE QUALITY CHECK

ARTISANHUB

GROUP MEMBERS:

NAMRA IMTIAZ: SE-22060

URWA QADIR : SE-22061

FATIMA HASSAN: SE-22098

JASSICA ALLEN: SE-22099

SUBMIED TO MISS SHIZA

1) Introduction

Project Overview:

This project is a MERN (MongoDB, Express, React, Node.js) stack-based e-commerce application. The application allows users to browse products, add items to their cart, manage their shopping, and securely complete purchases. It features separate admin and user login pages, product listings, and a shopping cart system. Admins can manage product listings (add, update, delete) and users can browse, filter, and securely checkout their selections.

2) Problems Encountered While Coding the Project

1. **Inconsistent Coding Standards:**
   Since more than one member was handling different concurrent modules of the project, there were variations in coding styles among team members. Impacting the code readability and an increase in likelihood of bugs.

2. **Lack of Documentation**
   No up-to-dated documents were made to keep the track in code changes and as a result more time was consumed in code debugging and adding additional features.

3. **Unhandled Errors:**
   The project implements a global error handler in Express to manage unexpected errors, and error boundaries in React to catch UI-related errors gracefully.

4. **Performance Issues:**
   Identified performance bottlenecks, such as unoptimized database queries, were addressed by creating indexes in MongoDB and using pagination for large datasets.

5. **Continuous Integration and Deployment:**
   Due the lack in pair programming, integrating and compiling the code in one place was an issue. Deploying the code in continuous format was not done as expected.

6. **Code Duplication**
   Same coding logic was implemented for other different modules, increasing the time in code maintenance, making test cases and adding unnecessary middlewares for security concerns.

7. **Code Smells**
   Different types of code smells were detected, that were causing an issue in reducing nesting levels, and consolidating redundant code. Which was later refactored after the code quality analysis scan.

3) Strategies for Maintaining Code Quality

Code Quality Check using SonarCloud

Overview of SonarCloud:
SonarCloud is a powerful cloud-based tool for tracking code quality, highlighting potential issues like code smells, bugs, vulnerabilities, and hotspots. It provides comprehensive insights into code maintainability and detects areas that could affect performance or security. By integrating SonarCloud, our team benefits from regular, automated quality checks that offer actionable feedback on the codebase.

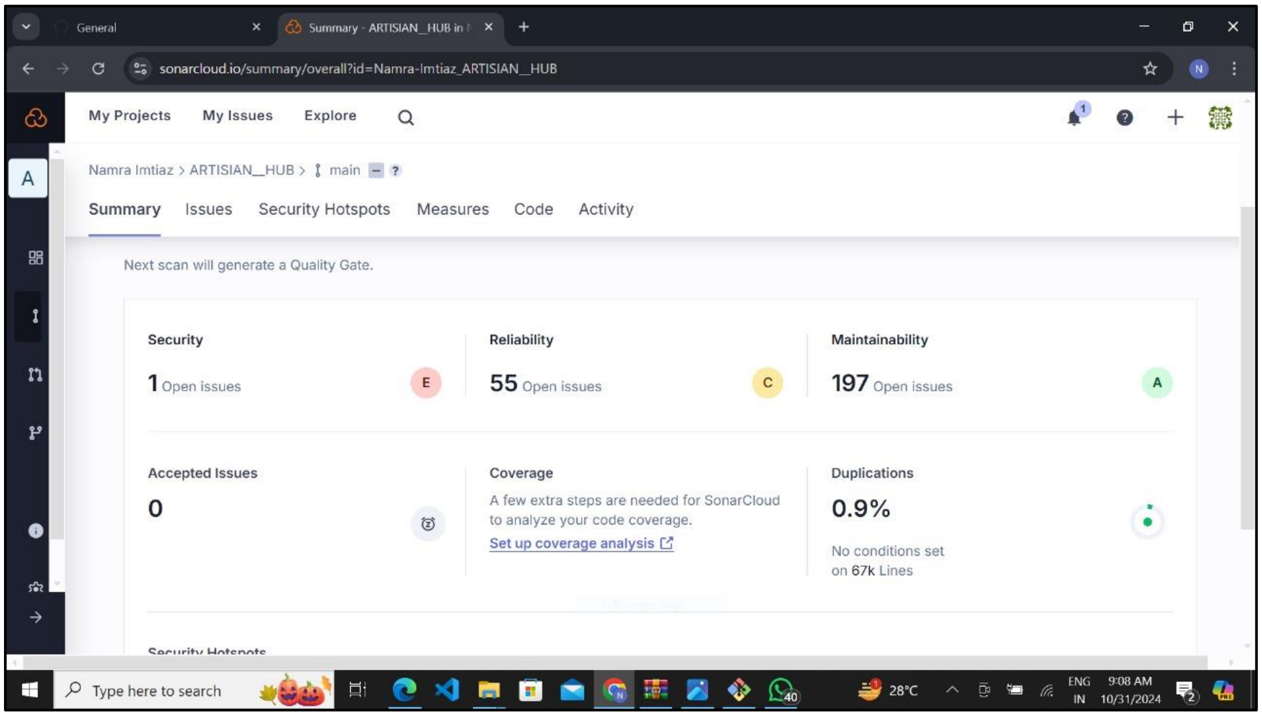
REVIEWING THE ANALYSIS RESULTS

- **Reliability**: Examine identified bugs, assess their impact on functionality, and prioritize fixes.

- **Security:** Inspect flagged vulnerabilities, understand potential security risks, and review recommended remediation actions.

- **Maintainability:** Analyze detected code smells, which highlight areas to improve readability, simplify complexity, and enhance future maintainability.

- **Coverage and Duplications:** Evaluate test coverage to ensure code is well-tested, and review duplicate code sections that may need consolidation for efficiency.


General
Summary - ARTISIAN__HUB in
sonarcloud.io/summary/overall?id=Namra-Imtiaz_ARTISIAN__HUB
My Projects
My Issues
Explore
Namra Imtiaz > ARTISIAN__HUB > main
Summary
Issues
Security Hotspots
Measures
Code
Activity

Next scan will generate a Quality Gate.

Security
1 Open issues E

Reliability
55 Open issues C

Maintainability
197 Open issues A

Accepted Issues
0

Coverage
A few extra steps are needed for SonarCloud to analyze your code coverage.
Set up coverage analysis

Duplications
0.9%
No conditions set on 67k Lines

Security Hotspots

4) Handling Issues Encountered in Code Quality Scan

1. Prioritization and Addressing Critical Bugs:

Bug triaging was conducted to categorize issues based on their severity and impact. Security-related bugs and those affecting core functionality were prioritized, followed by performance issues and cosmetic bugs. Addressing high-priority bugs first helps maintain a secure and reliable application.

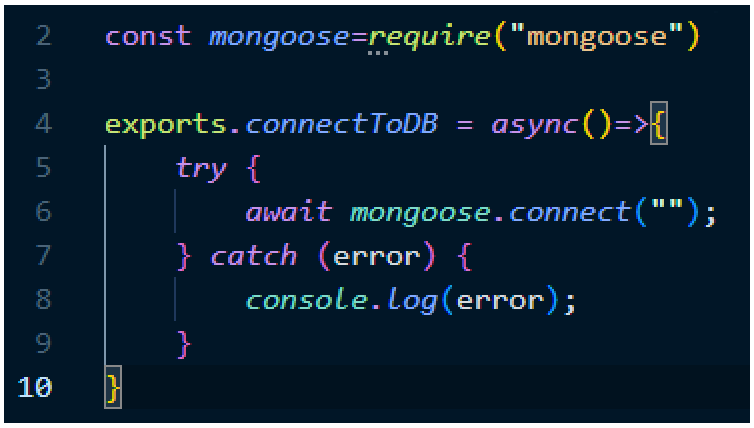
2. Refactoring Code Smells:

Refactoring involved breaking down complex functions, reducing nesting levels, and consolidating redundant code. This improved readability and made the codebase more manageable. The goal was to follow SOLID principles to make the code more modular, enhancing maintainability.

Following code snippets were prioritized based on their impact on whole project execution:

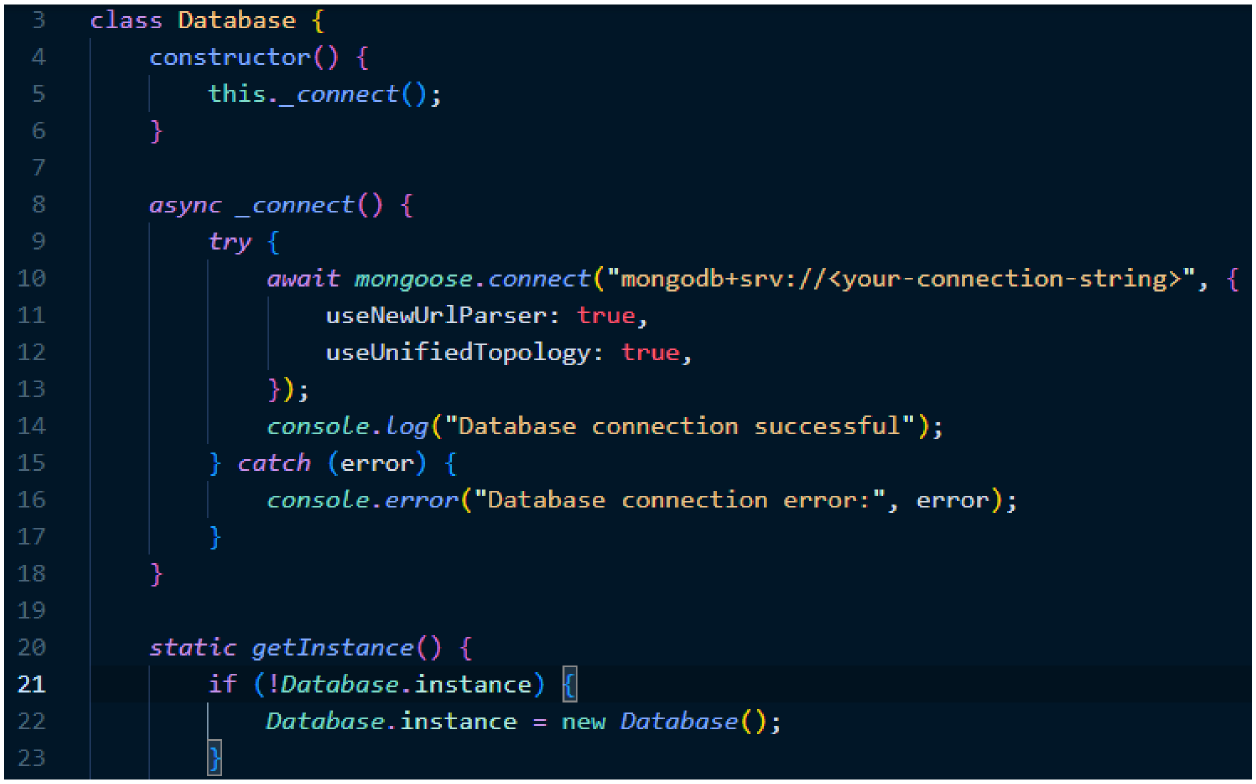
2.1 Refactoring Database Connection Module By Implementing Singleton Design Pattern

Before Refactoring:



```
const mongoose=require("mongoose")

exports.connectToDB = async()=>{
    try {
        await mongoose.connect("");
    } catch (error) {
        console.log(error);
    }
}
```

After Refactoring:



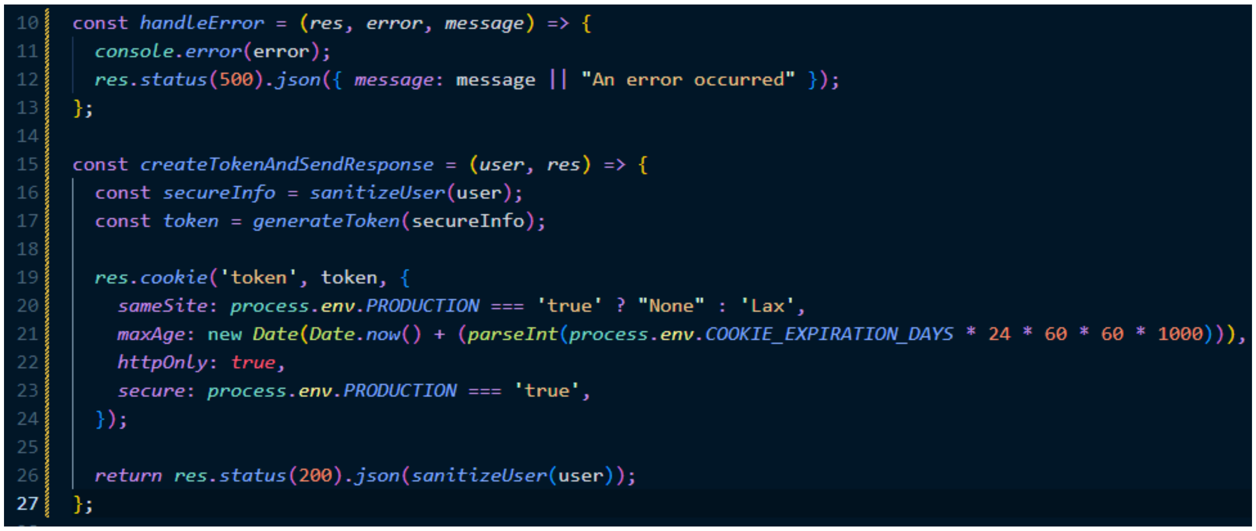
```
class Database {
    constructor() {
        this._connect();
    }

    async _connect() {
        try {
            await mongoose.connect("mongodb+srv://<your-connection-string>", {
                useNewUrlParser: true,
                useUnifiedTopology: true,
            });
            console.log("Database connection successful");
        } catch (error) {
            console.error("Database connection error:", error);
        }
    }

    static getInstance() {
        if (!Database.instance) {
            Database.instance = new Database();
        }
```

A try catch block is used to handle the exception in class instance.

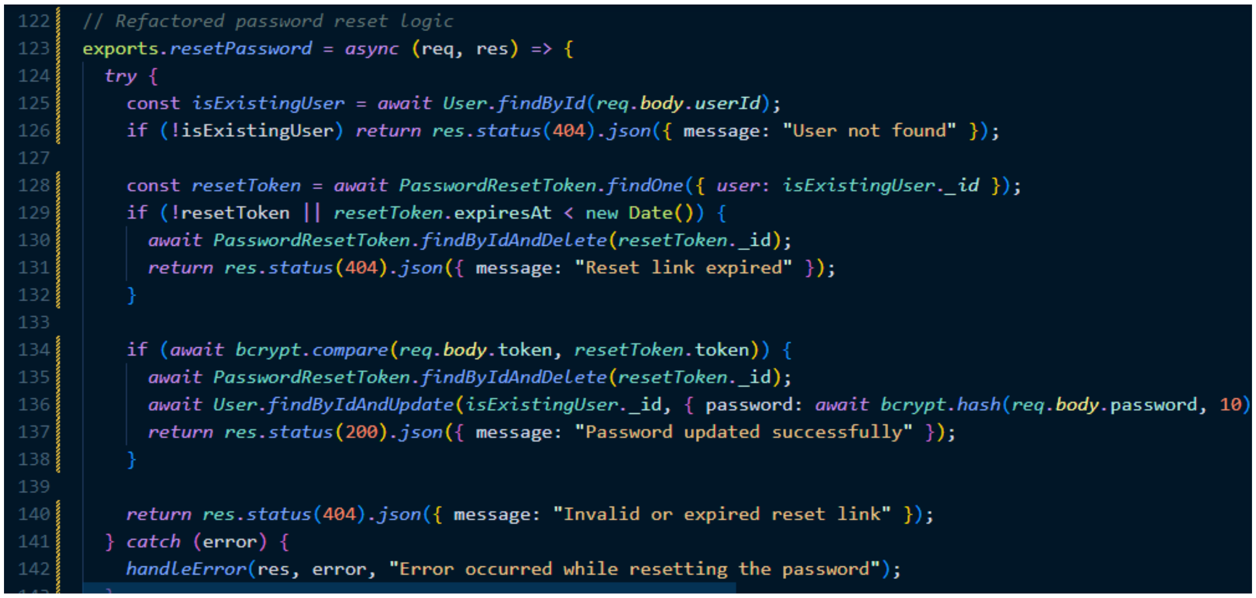
2.2 Refactoring Authentication Logic By Implementing Code Reusability Strategy:

Helper Functions: Functions like handleError and createTokenAndSendResponse reduce code repetition and centralize error handling and response logic.
Error Handling: Instead of handling errors in multiple places, a single function (handleError) is used to log and send a consistent response for errors.
Token Creation: A reusable function (createTokenAndSendResponse) handles the process of creating a JWT token and sending it in the response, ensuring the same behavior for both signup and login.

Refactored Code:



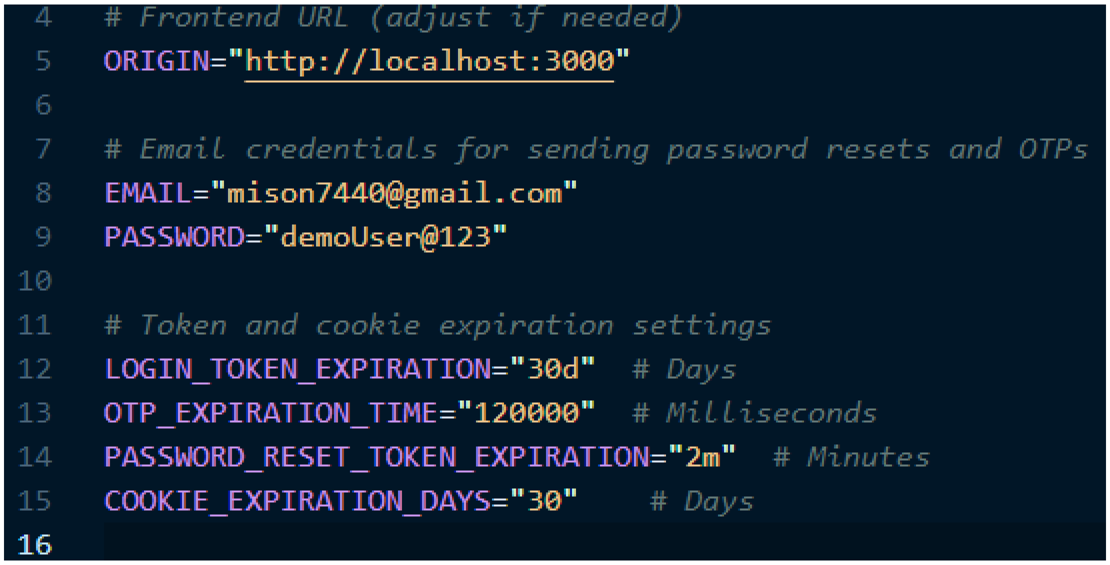
```javascript
const handleError = (res, error, message) => {
  console.error(error);
  res.status(500).json({ message: message || "An error occurred" });
};

const createTokenAndSendResponse = (user, res) => {
  const secureInfo = sanitizeUser(user);
  const token = generateToken(secureInfo);

  res.cookie('token', token, {
    sameSite: process.env.PRODUCTION === 'true' ? "None" : 'Lax',
    maxAge: new Date(Date.now() + (parseInt(process.env.COOKIE_EXPIRATION_DAYS * 24 * 60 * 60 * 1000))),
    httpOnly: true,
    secure: process.env.PRODUCTION === 'true',
  });

  return res.status(200).json(sanitizeUser(user));
};
```



```javascript
// Refactored password reset logic
exports.resetPassword = async (req, res) => {
  try {
    const isExistingUser = await User.findById(req.body.userId);
    if (!isExistingUser) return res.status(404).json({ message: "User not found" });

    const resetToken = await PasswordResetToken.findOne({ user: isExistingUser._id });
    if (!resetToken || resetToken.expiresAt < new Date()) {
      await PasswordResetToken.findByIdAndDelete(resetToken._id);
      return res.status(404).json({ message: "Reset link expired" });
    }

    if (await bcrypt.compare(req.body.token, resetToken.token)) {
      await PasswordResetToken.findByIdAndDelete(resetToken._id);
      await User.findByIdAndUpdate(isExistingUser._id, { password: await bcrypt.hash(req.body.password, 10)
      return res.status(200).json({ message: "Password updated successfully" });
    }

    return res.status(404).json({ message: "Invalid or expired reset link" });
  } catch (error) {
    handleError(res, error, "Error occurred while resetting the password");
```

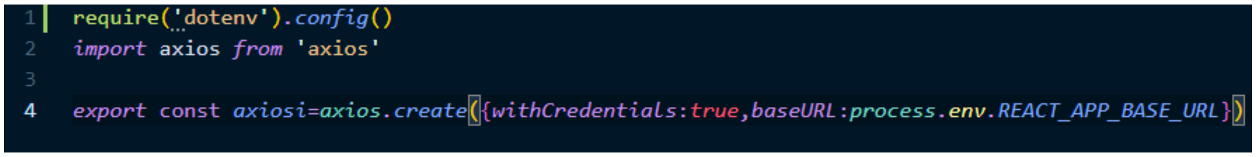
2.3 Securing Hotspots by Using Environmental Variables:

Storing sensitive information in environment variables using .env files and load them securely with dotenv.



```
# Frontend URL (adjust if needed)
ORIGIN="http://localhost:3000"

# Email credentials for sending password resets and OTPs
EMAIL="mison7440@gmail.com"
PASSWORD="demoUser@123"

# Token and cookie expiration settings
LOGIN_TOKEN_EXPIRATION="30d"  # Days
OTP_EXPIRATION_TIME="120000"  # Milliseconds
PASSWORD_RESET_TOKEN_EXPIRATION="2m"  # Minutes
COOKIE_EXPIRATION_DAYS="30"    # Days
```

Can be accessed by:



```
require('dotenv').config()
import axios from 'axios'

export const axiosi=axios.create({withCredentials:true,baseURL:process.env.REACT_APP_BASE_URL})
```

2.4 For Keeping a Coding Standard Prettier Linting Tool was Used:

Code Formatters: Prettier was used to enforce consistent formatting and styling, reducing cognitive load for team members and keeping the codebase clean.

2.5 Continuous Integration/Continuous Deployment (CI/CD):

A CI/CD pipeline was set up with GitHub Actions to automatically test and deploy changes, ensuring that each merge is stable and minimizes human error.

2.6 Testing :

Manual testing was conducted for critical features to ensure the validation of user flows.

2.7 Code Duplication:

Code duplication was reduced by modularizing common functionality, such as form validation and API calls, into reusable components and functions.

5) Tools and Technologies Used

SonarCloud: For code quality monitoring and bug detection.

Prettier: For linting and formatting, ensuring a consistent coding style.

MongoDB Compass: For database visualization and management, especially for monitoring query performance and optimizing indexes.

6) Conclusion

- **Summary of Key Findings and Strategies**:

SonarCloud helped identify critical bugs, security hotspots, and areas for improvement in maintainability. Refactoring, improving test coverage, and code reviews were effective in improving code quality.

- **Recommendations for Future Improvements:**

Suggest incorporating additional automated security testing and increasing the depth of end-to-end testing.

- **Importance of Ongoing Code Quality Maintenance:**

Code quality is a continuous process. Regular maintenance and monitoring are recommended to sustain the app’s performance, security, and scalability.