Full Stack Development with MERN

API Development and Integration Report

Date	21/Jul/2024
Team ID	SWTID1720103759
Project Name	Resolve Radar
Maximum Marks	

Project Title: Complaint Management System
Date: 21st July 2024
Prepared by: Resolve Radar

Objective

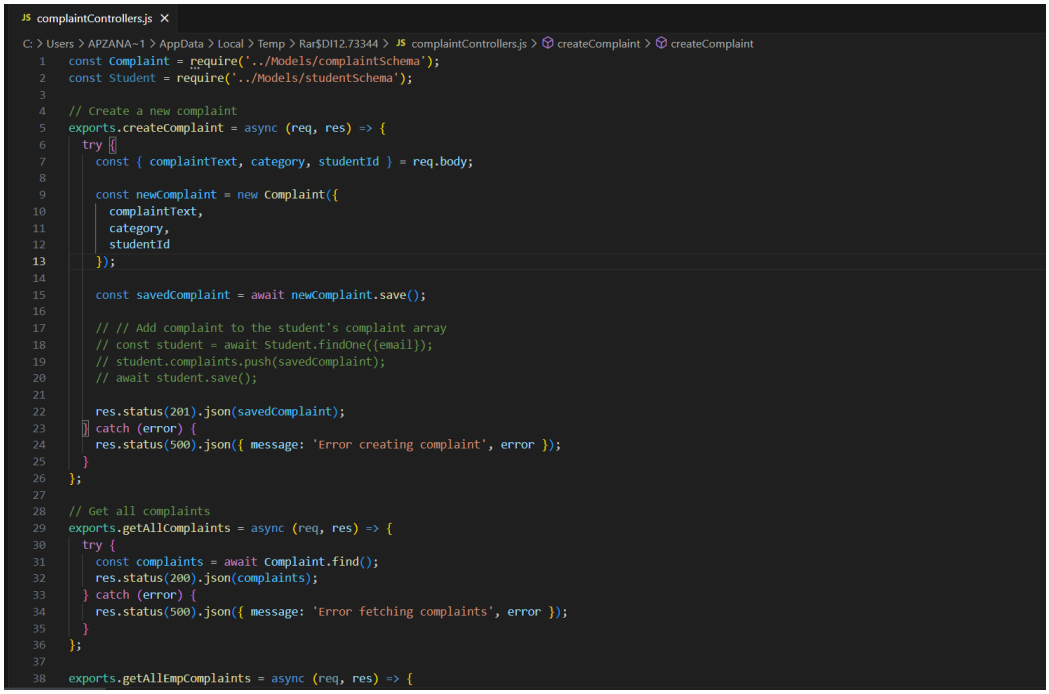
The objective of this report is to document the API development progress and key aspects of the backend services implementation for the Complaint Management System project.

Technologies Used

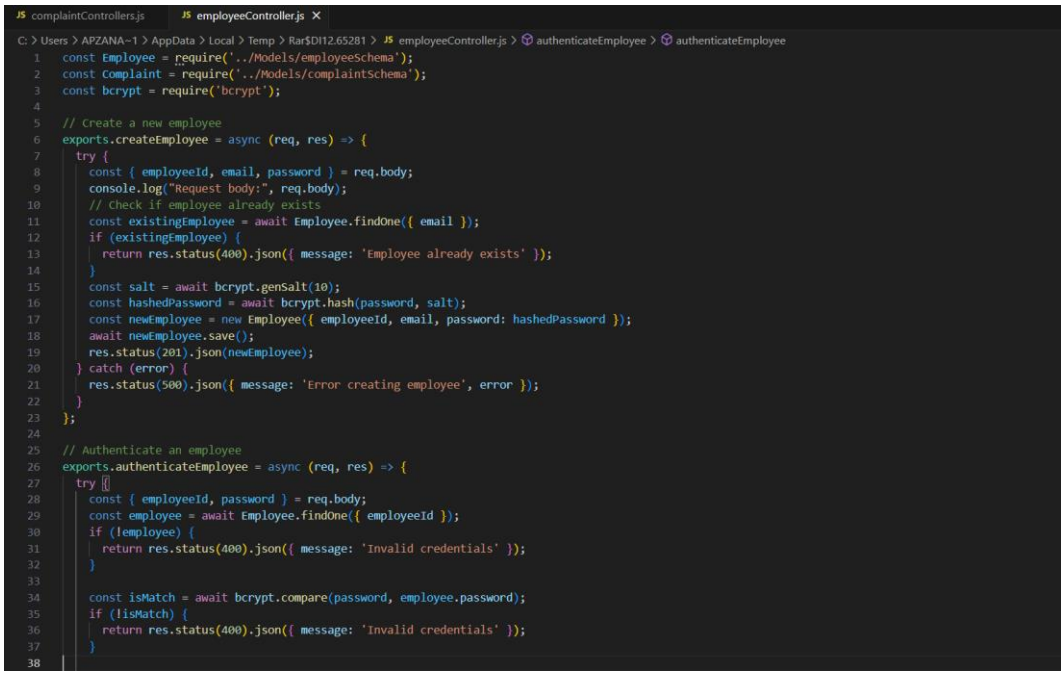
- **Backend Framework:** Node.js with Express.js
- **Database:** MongoDB
- **Authentication:**

Project Structure

- **Complaint Controller**

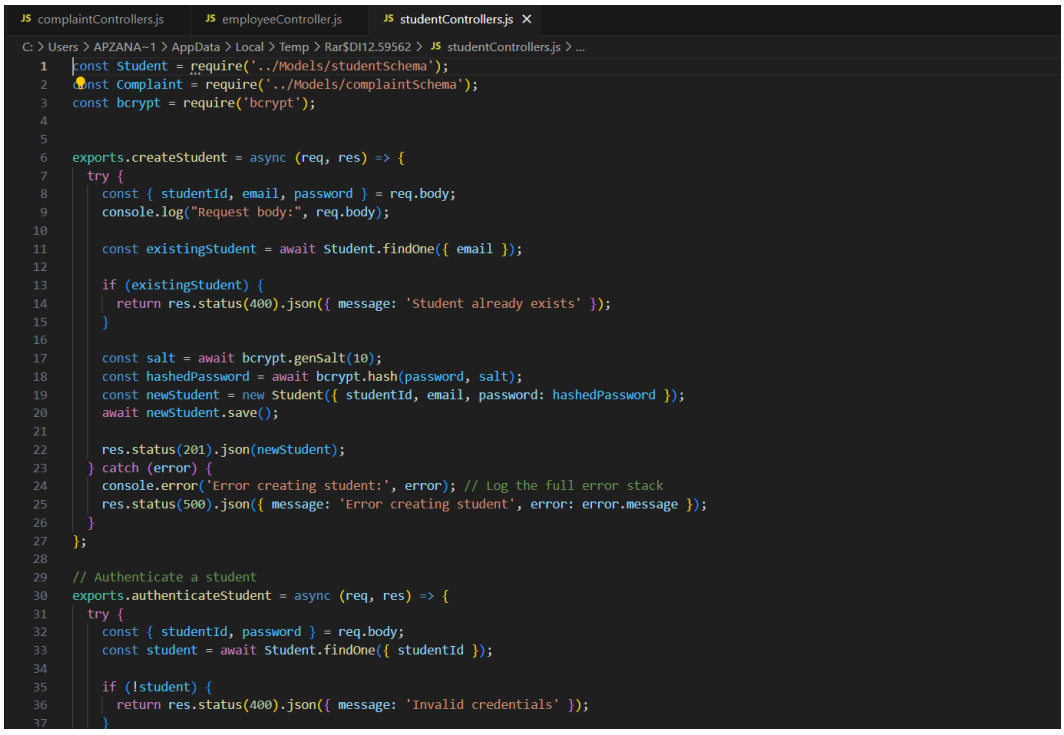


- **Employee Controller**



```
const Employee = require('../Models/employeeSchema');
const Complaint = require('../Models/complaintSchema');
const bcrypt = require('bcrypt');

// Create a new employee
exports.createEmployee = async (req, res) => {
  try {
    const { employeeId, email, password } = req.body;
    console.log("Request body:", req.body);
    // Check if employee already exists
    const existingEmployee = await Employee.findOne({ email });
    if (existingEmployee) {
      return res.status(400).json({ message: 'Employee already exists' });
    }
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
    const newEmployee = new Employee({ employeeId, email, password: hashedPassword });
    await newEmployee.save();
    res.status(201).json(newEmployee);
  } catch (error) {
    res.status(500).json({ message: 'Error creating employee', error });
  }
};

// Authenticate an employee
exports.authenticateEmployee = async (req, res) => {
  try {
    const { employeeId, password } = req.body;
    const employee = await Employee.findOne({ employeeId });
    if (!employee) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }

    const isMatch = await bcrypt.compare(password, employee.password);
    if (!isMatch) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }
```

- **Student Controller**



```
const Student = require('../Models/studentSchema');
const Complaint = require('../Models/complaintSchema');
const bcrypt = require('bcrypt');


exports.createStudent = async (req, res) => {
  try {
    const { studentId, email, password } = req.body;
    console.log("Request body:", req.body);

    const existingStudent = await Student.findOne({ email });

    if (existingStudent) {
      return res.status(400).json({ message: 'Student already exists' });
    }

    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
    const newStudent = new Student({ studentId, email, password: hashedPassword });
    await newStudent.save();

    res.status(201).json(newStudent);
  } catch (error) {
    console.error('Error creating student:', error); // Log the full error stack
    res.status(500).json({ message: 'Error creating student', error: error.message });
  }
};

// Authenticate a student
exports.authenticateStudent = async (req, res) => {
  try {
    const { studentId, password } = req.body;
    const student = await Student.findOne({ studentId });

    if (!student) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }
```

- **Complaint Schema**

```
JS complaintControllers.js | JS employeeController.js | JS studentControllers.js | JS complaintSchema.js ×
C: > Users > APZANA~1 > AppData > Local > Temp > Rar$DI13.91062 > JS complaintSchema.js > ...
1  const mongoose = require('mongoose');
2
3
4  const complaintSchema = new mongoose.Schema({
5      complaintText: { type: String, required: true },
6      category: { type: String, required: true },
7      date: { type: Date, default: () => new Date().toISOString().split('T')[0] },
8      studentId: { type: String, required: true },
9      status: {type:String, default:'Pending'},
10     feedback: {type:String, default: 'No feedback' }  // Assuming studentId is a string identifier
11 }, { versionKey: false });
12
13 const Complaint = mongoose.model('Complaint', complaintSchema);
14
15 module.exports = Complaint;
16
```

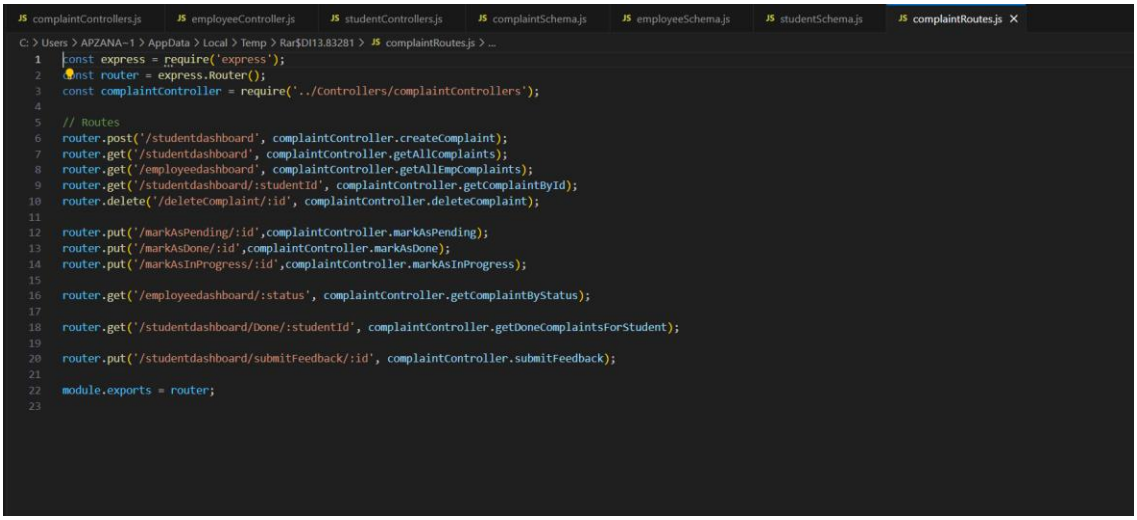
- **Employee Schema**

```
JS complaintControllers.js | JS employeeController.js | JS studentControllers.js | JS complaintSchema.js | JS employeeSchema.js ×
C: > Users > APZANA~1 > AppData > Local > Temp > Rar$DI13.72828 > JS employeeSchema.js > ...
1  // models/employeeSchema.js
2
3  const mongoose = require('mongoose');
4
5  const employeeSchema = new mongoose.Schema({
6    employeeId: {type: String, required: true, unique: true},
7    email: {type: String, required: true, unique: true},
8    password: {type: String, required: true,},
9    // complaints: [{
10   //   type: mongoose.Schema.Types.ObjectId,
11   //   ref: 'Complaint',
12   // }],
13 }, {versionKey:false} );
14 employeeSchema.index({ roll: 1 }, { sparse: true });
15 const Employee = mongoose.model('Employee', employeeSchema);
16 module.exports = Employee;
17
18
19 //{ timestamps: true }
```

- **Student Schema**

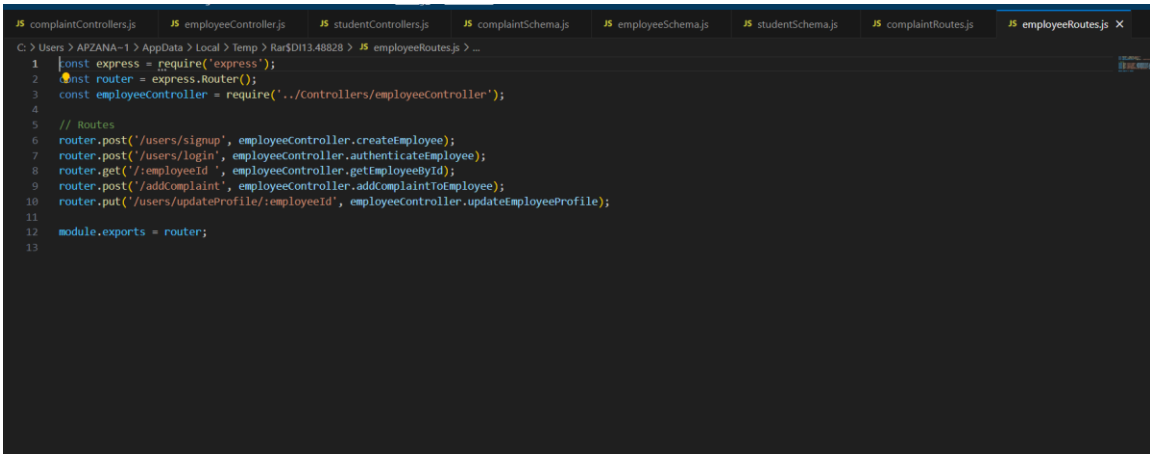
```
JS complaintControllers.js | JS employeeController.js | JS studentControllers.js | JS complaintSchema.js | JS employeeSchema.js | JS studentSchema.js ×
C: > Users > APZANA~1 > AppData > Local > Temp > Rar$DI13.87562 > JS studentSchema.js > ...
1  const mongoose = require('mongoose');
2  // Define the Student Schema
3  const studentSchema = new mongoose.Schema({
4      studentId: { type: String, required: true, unique: true },
5      email: { type: String, required: true, unique: true },
6      password: { type: String, required: true },
7      // complaints: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Complaint' }] // Array of complaint references
8  }, { versionKey: false });
9  studentSchema.index({ roll: 1 }, { sparse: true });
10 // Create and export the Student model
11 const Student = mongoose.model('Student', studentSchema);
12
13 module.exports = Student;
14
```

- **Complaint Routes**



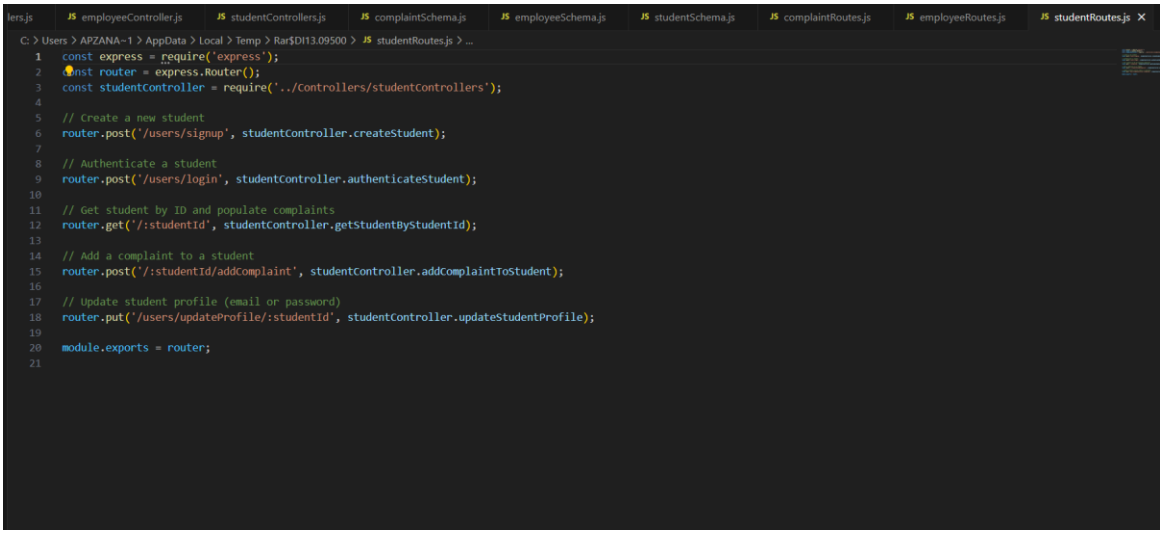
```
const express = require('express');
const router = express.Router();
const complaintController = require('../Controllers/complaintControllers');

// Routes
router.post('/studentdashboard', complaintController.createComplaint);
router.get('/studentdashboard', complaintController.getAllComplaints);
router.get('/employeedashboard', complaintController.getAllEmpComplaints);
router.get('/studentdashboard/:studentId', complaintController.getComplaintById);
router.delete('/deleteComplaint/:id', complaintController.deleteComplaint);

router.put('/markAsPending/:id',complaintController.markAsPending);
router.put('/markAsDone/:id',complaintController.markAsDone);
router.put('/markAsInProgress/:id',complaintController.markAsInProgress);

router.get('/employeedashboard/:status', complaintController.getComplaintByStatus);

router.get('/studentdashboard/Done/:studentId', complaintController.getDoneComplaintsForStudent);

router.put('/studentdashboard/submitFeedback/:id', complaintController.submitFeedback);

module.exports = router;
```

- **Employee Routes**



```
const express = require('express');
const router = express.Router();
const employeeController = require('../Controllers/employeeController');

// Routes
router.post('/users/signup', employeeController.createEmployee);
router.post('/users/login', employeeController.authenticateEmployee);
router.get('/:employeeId ', employeeController.getEmployeeById);
router.post('/addComplaint', employeeController.addComplaintToEmployee);
router.put('/users/updateProfile/:employeeId', employeeController.updateEmployeeProfile);

module.exports = router;
```

- **Student Routes**



```
const express = require('express');
const router = express.Router();
const studentController = require('../Controllers/studentControllers');

// Create a new student
router.post('/users/signup', studentController.createStudent);

// Authenticate a student
router.post('/users/login', studentController.authenticateStudent);

// Get student by ID and populate complaints
router.get('/:studentId', studentController.getStudentByStudentId);

// Add a complaint to a student
router.post('/:studentId/addComplaint', studentController.addComplaintToStudent);

// Update student profile (email or password)
router.put('/users/updateProfile/:studentId', studentController.updateStudentProfile);

module.exports = router;
```

Key Directories and Files

1. **/controllers**
    - Contains functions to handle requests and responses.
2. **/models**
    - Includes Mongoose schemas and models for MongoDB collections.
3. **/routes**
    - Defines the API endpoints and links them to controller functions.
4. **.env**
    - Configuration files for database connections, environment variables, etc.

API Endpoints:

User Authentication

- **POST /Express/users/signup:** Handles user signup**.**
- **POST /Express/users/login:** Handles user authentication.

User Management

- **GET /Express/studentId:** Retrieves the Student data for display by student ID.
- **GET /Express/employeeId:** Retrieves the Employee data for display by employee ID.
- **PUT /Express/users/updateProfile/studentId**: Update student profile (email/password).
- **PUT /Express/users/updateProfile/employeeId:** Update employee profile (email/password).

Complaint Management

- **POST /Express/studentId/addComplaint:** Add a complaint to a student.
- **POST /Express/studentdashboard:** Create complaints.
- **GET /Express/studentdashboard**: Retrieves all the complaints data.
- **GET /Express/studentdashboard/studentId:** Retrieves all the complaints data for display by student ID.
- **GET /Express/employeedashboard:** Retrieves all the Employee's complaints data.
- **GET /Express/employeedashboard/status:** Retrieves all the complaints data for display by complaint status.
- **PUT /Express/markAsDone/id:** Update the complaint status and mark it as done.
- **PUT /Express/markAsInProgress/id**: Update the complaint status and mark it as in progress

Integration with Frontend
The backend communicates with the frontend via RESTful APIs. Key points of integration include:

- **User Authentication:** Tokens are exchanged between the frontend and backend to manage user authentication securely. This ensures that only authenticated users can access protected resources and endpoints.
- **Data Fetching:** The frontend components interact with the backend to retrieve and display necessary data. This includes making API requests to fetch user profiles, view complaint details, and update user information. The frontend uses these API endpoints to send requests and receive responses, which are then processed to dynamically update the user interface. This ensures that the UI reflects real-time data and user actions.

Error Handling

- **Error Handling:** The application employs a centralized error-handling strategy using middleware. In the 'app.js' file, an error handling middleware is implemented to catch and handle errors occurring throughout the application. This middleware logs the error stack for debugging purposes and sends a generic '500 Internal Server Error' response to the client. This ensures that errors are managed consistently and allows for easier debugging and maintenance.

Security Considerations

- **Authentication:** The application uses secure token-based authentication for verifying user credentials. In the provided controller file, passwords are hashed using 'bcrypt' before storage to ensure that sensitive information is not exposed. During authentication, the provided password is compared with the hashed password stored in the database. Although token-based authentication (e.g., JWT) is not shown in the current implementation, it should be considered for secure and scalable user sessions.

- **Data Encryption:** Sensitive data is encrypted using 'bcrypt' for password hashing, ensuring that plaintext passwords are not stored in the database. Data encryption at rest and in transit is crucial for protecting sensitive information. While the current implementation focuses on password encryption, additional measures such as HTTPS should be used to secure data in transit, and encryption at rest should be implemented to protect stored data.