**~~Logger.js~~**

Your JavaScript code using Winston for logging looks solid. However, let’s review it for potential errors, improvements, and suggestions:

### Code Analysis

1. \*\*Error Checking:\*\*

- There are no obvious syntax errors or issues in the code. The use of Winston appears correct.

2. \*\*Improvement and Suggestions:\*\*

- \*\*Log Directory Handling:\*\* Ensure that the directory 'logs' exists. If it doesn't, logging to a file will fail.

```javascript

const fs = require('fs');

const path = require('path');

const logDir = 'logs';

if (!fs.existsSync(logDir)) {

fs.mkdirSync(logDir);

}

```

- \*\*Environment-Specific Log Levels:\*\* It is a good practice to differentiate log levels based on the environment (e.g., development, production).

```javascript

const logger = createLogger({

level: process.env.NODE\_ENV === 'production' ? 'warn' : 'debug', // Adjust log level based on the environment

```

- \*\*Add More Transports:\*\* Consider adding transports for different environments or uses, such as logging to external services in a production environment.

```javascript

if (process.env.NODE\_ENV === 'production') {

logger.add(new transports.Http({ host: 'log-server', port: 9000 }));

}

```

- \*\*Regular Log Rotation:\*\* For log files, consider adding log rotation to prevent single files from becoming excessively large.

```

const { createLogger, format, transports } = require('winston');

require('winston-daily-rotate-file');

const fileTransport = new transports.DailyRotateFile({

filename: 'logs/app-%DATE%.log',

datePattern: 'YYYY-MM-DD',

zippedArchive: true,

maxSize: '20m',

maxFiles: '14d'

});

const logger = createLogger({

transports: [

new transports.Console(),

fileTransport

]

});

```

3. \*\*Security Considerations:\*\*

- Be cautious about logging sensitive information, such as passwords or API keys. Ensure this type of information isn’t inadvertently logged.

### Final Adjusted Code Example:

```javascript

// logger.js

const fs = require('fs');

const path = require('path');

const { createLogger, format, transports } = require('winston');

require('winston-daily-rotate-file');

const { combine, timestamp, printf, errors } = format;

const logDir = 'logs';

if (!fs.existsSync(logDir)) {

fs.mkdirSync(logDir);

}

// Custom format for the logs

const myFormat = printf(({ level, message, timestamp, stack }) => {

return `${timestamp} [${level}]: ${stack || message}`;

});

// Create a Winston logger instance

const logger = createLogger({

level: process.env.NODE\_ENV === 'production' ? 'warn' : 'debug', // Adjust log level based on the environment

format: combine(

timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),

errors({ stack: true }), // Log stack traces for errors

myFormat

),

transports: [

new transports.Console(), // Log to console

new transports.DailyRotateFile({

filename: 'logs/app-%DATE%.log',

datePattern: 'YYYY-MM-DD',

zippedArchive: true,

maxSize: '20m',

maxFiles: '14d'

})

]

});

if (process.env.NODE\_ENV === 'production') {

logger.add(new transports.Http({ host: 'log-server', port: 9000 }));

}

module.exports = logger;

```

This incorporates directory handling, environment-specific log levels, log file rotation, and the ability to add more transports for different logging needs.

**~~Server.js~~**

Your Express.js server setup looks comprehensive and well-structured, but there are always areas for improvement. Here are some suggestions, improvements, and potential error checks for your `server.js` file:

### Overall Suggestions

1. \*\*Environment Variable Checks\*\*:

- Check for the presence of all necessary environment variables at the start and fail early if any are missing.

2. \*\*Error Handling Middleware Placement\*\*:

- The error handling middleware should be placed at the end, as you correctly have done. However, adding some more specific middleware for known error types could improve the user experience.

3. \*\*Security Enhancements\*\*:

- Adding various security middlewares like `helmet` for setting various HTTP headers for security.

- Ensure that cookies are appropriately configured for security, e.g., `secure` and `httpOnly` attributes.

### Detailed Analysis

#### Environment Variable Check

```javascript

if (!process.env.DATABASE\_URL || !process.env.SESSION\_SECRET || !process.env.PORT) {

console.error("Error: Necessary environment variables not set. Please check your .env configuration file.");

process.exit(-1);

}

```

Additionally, you might want to warn or log specific missing environment variables for easier debugging.

#### Logger Configuration

- Ensure the log directory is writable; otherwise, the application might crash without proper file system permissions.

- Consider adding rotation to your log files to avoid infinite growth.

- For further improvement, you can include more log levels (error, warn, info, debug, etc.).

#### Mongoose Connection Options

- Adding more robust retry mechanisms and timeout configurations could enhance your application’s resilience.

```javascript

mongoose.connect(process.env.DATABASE\_URL, { useNewUrlParser: true, useUnifiedTopology: true, useCreateIndex: true })

.then(() => logger.info('Database connected successfully'))

.catch((err) => {

logger.error(`Database connection error: ${err.message}`, err);

process.exit(1);

});

```

#### Middleware Order

- While your middleware order seems correct, ensure that any rate limiting, security, or logging middleware is appropriately placed. For example, it's good practice to have `helmet`, a security middleware, just after other core middlewares.

#### Security Middleware

```javascript

const helmet = require('helmet');

app.use(helmet());

```

#### Sessions and Cookies

- While you correctly set `cookie: { secure: process.env.NODE\_ENV === 'production', httpOnly: true }`, ensure it aligns with your application's deployment and front-end requirements.

#### CSRF Protection

- CSRF protection is accurately placed, but ensure that API routes that don't require session handling are excluded if not using tokens.

#### Error Handling Middleware

- Your error handling middleware is good. For better user experience, you could improve the error messages or handle specific known errors like database errors, authentication errors, etc.

```javascript

app.use((err, req, res, next) => {

if (err.code === 'EBADCSRFTOKEN') {

logger.error(`CSRF token error: ${err.message}`, {

error: err,

sessionId: req.sessionID,

timestamp: new Date().toISOString(),

path: req.path,

method: req.method,

});

res.status(403).render('error', { message: 'Invalid CSRF token' });

} else {

logger.error(`Unhandled application error: ${err.message}`, {

error: err,

stack: err.stack,

sessionId: req.sessionID,

timestamp: new Date().toISOString(),

});

res.status(500).render('error', { message: 'Internal Server Error' });

}

});

```

#### Start Server

- Consider adding signal handlers to gracefully shut down the server on termination signals (SIGINT, SIGTERM).

```javascript

process.on('SIGINT', () => {

logger.info('SIGINT signal received: closing HTTP server');

server.close(() => {

logger.info('HTTP server closed');

process.exit(0);

});

});

const server = app.listen(port, () => {

logger.info(`Server running at http://localhost:${port}`);

});

```

Overall, your code is well-organized and adheres to solid practices. These improvements will help ensure your application is more secure, maintainable, and easier to debug.

**~~authController.js~~**

The `authController.js` code you've provided appears to be well-structured and adheres to best practices with respect to basic user authentication workflows. However, there are some improvements and suggestions that can be made to further enhance the security, maintainability, and clarity of your code. Here are some observations and suggestions:

### 1. Error Handling and User Feedback

1. \*\*Improving User Feedback on Errors:\*\*

- Sending raw error messages directly to the client in a production environment can be a security risk. Instead, use user-friendly messages and log the detailed errors server-side.

```js

res.status(500).send('An error occurred during registration');

```

2. \*\*Improving Error Logging:\*\*

- While logging errors, consider using a logging library like `winston` or `morgan` for better log management.

### 2. Password Requirements and Validation

1. \*\*Password Validation:\*\*

- Consider adding validation for the password to ensure it meets certain security requirements (e.g., minimum length, complexity).

```js

const Joi = require('joi');

const schema = Joi.object({

username: Joi.string().min(3).required(),

password: Joi.string().min(8).required()

});

const { error } = schema.validate(req.body);

if (error) {

return res.status(400).send(error.details[0].message);

}

```

### 3. Rate Limiting

1. \*\*Implement Rate Limiting:\*\*

- Use a library like `express-rate-limit` to prevent brute-force attacks against login endpoints.

```js

const rateLimit = require('express-rate-limit');

const loginLimiter = rateLimit({

windowMs: 15 \* 60 \* 1000, // 15 minutes

max: 20, // limit each IP to 20 requests per windowMs

message: 'Too many login attempts, please try again later'

});

router.post('/login', loginLimiter, csrfProtection, async (req, res) => {

// ... existing code ...

});

```

### 4. Security Headers and Sessions

1. \*\*Set Secure Cookies:\*\*

- Ensure that you are setting secure cookies and using HTTPOnly and SameSite attributes to protect against common attacks.

```js

app.use(session({

secret: 'yourSecret',

resave: false,

saveUninitialized: true,

cookie: { secure: true, httpOnly: true, sameSite: 'strict' }

}));

```

### 5. Refactor into Separate Middleware Files

1. \*\*Modularize Middleware:\*\*

- Moving different middleware functions to separate files can improve code organization, especially as your application grows.

```js

// middleware/csrfProtection.js

const csrf = require('csurf');

const csrfProtection = csrf({ cookie: true });

module.exports = csrfProtection;

```

```js

// authController.js

const csrfProtection = require('../middleware/csrfProtection');

```

### 6. Additional Security Measures

1. \*\*Account Lockout on Multiple Failed Attempts:\*\*

- Consider implementing account lockout mechanisms after a certain number of failed login attempts.

2. \*\*Multi-factor Authentication (MFA):\*\*

- For increased security, consider adding MFA to your authentication flow.

### Example of Improved Code:

Here's a refactored version of the `authController.js` file incorporating some of the suggestions:

```js

const express = require('express');

const User = require('../models/User');

const bcrypt = require('bcrypt');

const csrfProtection = require('../middleware/csrfProtection');

const rateLimit = require('express-rate-limit');

const router = express.Router();

// Setup rate limiting for login attempts

const loginLimiter = rateLimit({

windowMs: 15 \* 60 \* 1000, // 15 minutes

max: 20, // limit each IP to 20 requests per windowMs

message: 'Too many login attempts, please try again later'

});

router.get('/register', csrfProtection, (req, res) => {

res.render('register', { csrfToken: req.csrfToken() });

});

router.post('/register', csrfProtection, async (req, res) => {

try {

const { username, password } = req.body;

// Validate input

const schema = Joi.object({

username: Joi.string().min(3).required(),

password: Joi.string().min(8).required()

});

const { error } = schema.validate(req.body);

if (error) {

return res.status(400).send(error.details[0].message);

}

await User.create({ username, password });

res.redirect('/auth/login');

} catch (error) {

console.error('Registration error:', error);

res.status(500).send('An error occurred during registration');

}

});

router.get('/login', csrfProtection, (req, res) => {

res.render('login', { csrfToken: req.csrfToken() });

});

router.post('/login', loginLimiter, csrfProtection, async (req, res) => {

try {

const { username, password } = req.body;

const user = await User.findOne({ username });

if (!user) {

console.warn(`User not found: ${username}`);

return res.status(401).send('Invalid username or password');

}

const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) {

console.warn(`Invalid password for user: ${username}`);

return res.status(401).send('Invalid username or password');

}

req.session.userId = user.\_id;

res.redirect('/');

} catch (error) {

console.error('Login error:', error);

res.status(500).send('An error occurred during login');

}

});

router.get('/logout', (req, res) => {

req.session.destroy(err => {

if (err) {

console.error('Error during session destruction:', err);

return res.status(500).send('Error logging out');

}

res.redirect('/auth/login');

});

});

module.exports = router;

```

These changes aim to improve the overall robustness, security, and maintainability of the code.

**~~clientController.js~~**

The provided code looks quite solid in terms of functionality and error handling. However, there are still some areas where improvements can be made. Here are a few suggestions:

1. \*\*Validation\*\*: Validate the input data in the `createClient` and `updateClient` functions to ensure that all required fields are present and valid.

2. \*\*Consistency in Error Responses\*\*: Make sure error responses are consistent throughout the code. For example, in some cases, the error response includes a `success: false` field, while in others, it only includes a `message`.

3. \*\*Status Codes\*\*: Use appropriate status codes for different types of errors. For instance, a `400 - Bad Request` is used for validation errors or malformed requests, while `500 - Internal Server Error` should be reserved for unexpected server-side errors.

4. \*\*DRY Principle\*\*: Avoid repetition by creating a utility function for logging errors and sending error responses.

5. \*\*Schema Validation for MongoDB\*\*: While not directly related to the provided code, make sure your MongoDB schema includes validation rules where possible.

6. \*\*Refactoring\*\*: Break down the code into smaller, more manageable chunks.

Here's the refactored version of `clientController.js` with the suggested improvements:

```javascript

const Client = require('../models/clientModel');

const { v4: uuidv4 } = require('uuid');

const logger = require('../logger'); // Import the logger

// Utility function for handling errors

const handleError = (res, error, message, statusCode = 500) => {

logger.error(message, error);

res.status(statusCode).json({

success: false,

message,

error: error.message

});

};

// Create new Client

exports.createClient = async (req, res) => {

try {

const { companyName, name, email, phoneNumber, streetAddress, city, state, zip } = req.body;

// Basic validation

if (!companyName || !name || !email || !phoneNumber || !streetAddress || !city || !state || !zip) {

return res.status(400).json({ message: 'All fields are required' });

}

const clientData = {

clientId: uuidv4(),

companyName,

name,

email,

phoneNumber,

streetAddress,

city,

state,

zip

};

const newClient = await Client.create(clientData);

logger.info(`New client created: ${newClient.name}`);

res.status(201).json({

success: true,

data: newClient

});

} catch (error) {

handleError(res, error, 'Error creating new client', 400);

}

};

// Search Clients

exports.searchClients = async (req, res) => {

try {

const query = req.query.query;

const clients = await Client.find({ name: { $regex: query, $options: 'i' } });

logger.info(`Clients search results for query: ${query}`);

res.status(200).json(clients);

} catch (error) {

handleError(res, error, 'Error searching clients', 500);

}

};

// Get Client by ID

exports.getClientById = async (req, res) => {

try {

const client = await Client.findById(req.params.id);

if (!client) {

return res.status(404).json({ success: false, message: 'Client not found' });

}

res.status(200).json({ success: true, data: client });

} catch (error) {

handleError(res, error, `Error fetching client with ID ${req.params.id}`, 500);

}

};

// Update Client

exports.updateClient = async (req, res) => {

const clientId = req.params.id;

try {

const updatedClient = await Client.findByIdAndUpdate(clientId, req.body, { new: true });

if (!updatedClient) {

return res.status(404).json({ success: false, message: 'Client not found' });

}

logger.info(`Client updated successfully: ${updatedClient.name}`);

res.status(200).json({ success: true, message: 'Client updated successfully', data: updatedClient });

} catch (error) {

handleError(res, error, `Error updating client with ID ${clientId}`, 400);

}

};

// Delete Client

exports.deleteClient = async (req, res) => {

const clientId = req.params.id;

try {

const client = await Client.findByIdAndDelete(clientId);

if (!client) {

return res.status(404).json({ success: false, message: 'Client not found' });

}

logger.info(`Client deleted successfully: ${client.name}`);

res.status(200).json({ success: true, message: 'Client deleted successfully' });

} catch (error) {

handleError(res, error, `Error deleting client with ID ${clientId}`, 500);

}

};

```

Key Changes:

1. Added a utility function `handleError` to handle error logging and responses consistently.

2. Included basic validation for required fields in `createClient`.

3. Made error responses and status codes more consistent.

4. Reused logic to reduce repetition and enhance maintainability.

**~~quoteController~~.js**

The provided JavaScript code (specifically for an Express.js application) is structured well, with good use of async/await, error handling, and logging. Here’s a detailed review of the code, highlighting potential issues and suggesting improvements:

### Issues and Suggested Improvements

1. \*\*Authentication Middleware Usage\*\*

- You’ve imported the `isAuthenticated` middleware, but it is not being used in any of the routes. Ensure you add this middleware to routes where authentication is required.

```javascript

const { isAuthenticated } = require('../middleware/authMiddleware');

```

2. \*\*Consistent Tax Rate Handling\*\*

- Directly using `7.5` as the default tax rate multiple times may cause issues if you decide to change the default tax rate later. Consider using a constant.

```javascript

const DEFAULT\_TAX\_RATE = 7.5;

// Replace all 7.5 occurrences with DEFAULT\_TAX\_RATE

const calculatedTaxRate = parseFloat(taxRate) || DEFAULT\_TAX\_RATE;

const taxRate = parseFloat(req.body.taxRate) || DEFAULT\_TAX\_RATE;

```

3. \*\*Error Handling\*\*

- Ensure usage of specific HTTP status codes in error handling. Consider 400 for validation errors and 500 for server errors.

```javascript

if (!Array.isArray(serviceItems) || serviceItems.length === 0) {

logger.error('Invalid serviceItems data provided.');

return res.status(400).json({ message: 'Invalid serviceItems data provided.' });

}

catch (error) {

logger.error(`Error calculating distance: ${error.message}`, error);

res.status(500).json({ message: 'Failed to calculate distance', error: error.message });

}

```

4. \*\*Consistent Logger Messages\*\*

- Ensure logging messages, especially errors, are consistent and provide necessary context.

```javascript

logger.error(`Error updating quote with ID ${req.params.id}: ${error.message}`, error);

```

5. \*\*Validation of Input Data\*\*

- Better validate user input data, for instance, ensuring that required fields are present.

- Adding validation before database queries can save unnecessary load.

```javascript

const { clientId, title, serviceItems } = req.body;

if (!clientId || !title || !serviceItems) {

return res.status(400).json({ message: 'Required fields are missing.' });

}

```

6. \*\*Improved Distinctiveness of Error Messages\*\*

- Improved distinctiveness in logs and error messages can aid in debugging: consider adding specific details.

7. \*\*Optimize Imports\*\*

- Remove any unused imports. For instance, `isAuthenticated` if it’s not used.

### Additional Suggestions

1. \*\*Use of Promises in Array Operations\*\*

- Your usage of `Promise.all()` in handling the service items is good. Consider wrapping the logic in a separate function for clarity.

```javascript

const processServiceItems = async (items) => {

return Promise.all(items.map(async (item) => {

const serviceItem = await ServiceItem.findById(item.serviceItemId);

if (!serviceItem) {

throw new Error(`Service item not found with ID: ${item.serviceItemId}`);

}

return {

serviceItemId: item.serviceItemId,

quantity: item.quantity,

customPrice: item.customPrice ? item.customPrice : serviceItem.price

};

}));

};

```

2. \*\*Consider Using a Schema Validation Library\*\*

- Libraries like Joi or express-validator can provide better data validation mechanisms.

```javascript

const Joi = require('joi');

const quoteSchema = Joi.object({

clientId: Joi.string().required(),

title: Joi.string().required(),

// Add rest of the fields with respective validations

});

const { error } = quoteSchema.validate(req.body);

if (error) {

return res.status(400).json({ message: error.details[0].message });

}

```

3. \*\*File Handling and Validation\*\*

- Ensure you validate file uploads properly, checking file types and sizes as necessary.

```javascript

const attachments = req.files?.['attachments'] ?? [];

const contracts = req.files?.['contracts'] ?? [];

if (attachments.some(file => !allowedTypes.includes(file.mimetype))) {

return res.status(400).json({ message: 'Invalid attachment file type.' });

}

if (contracts.some(file => !allowedTypes.includes(file.mimetype))) {

return res.status(400).json({ message: 'Invalid contract file type.' });

}

```

By incorporating these suggestions and improvements, you can make the code more robust, maintainable, and easier to debug.

**~~serviceItemController~~.js**

The provided JavaScript (Express.js) code manages a `ServiceItem` model with typical CRUD operations. Here are some points of feedback, improvements, and suggestions:

### Error Handling

1. \*\*Consistent Error Messages\*\*

Use consistent error messages to help with debugging and ensure a good user experience.

```javascript

res.status(500).json({

success: false,

error: error.message // Maintain consistency

});

```

2. \*\*Error Codes\*\*

Ensure error status codes are consistent and indicative of the actual error.

```javascript

// For internal server errors

res.status(500).json({

success: false,

error: 'Internal server error'

});

// For bad requests (e.g., validation errors)

res.status(400).json({

success: false,

error: 'Bad request'

});

```

### Logging

1. \*\*Avoid Excessive Logs in Production\*\*

Limit logging details in production environments to avoid performance issues and potential security risks.

### Validation

1. \*\*Input Validation\*\*

Consider adding validation logic for input data before database operations. Libraries like `Joi` or middlewares like `express-validator` can be used.

```javascript

// Example with express-validator

const { body, validationResult } = require('express-validator');

exports.createServiceItem = [

body('name').not().isEmpty().withMessage('Name is required'),

async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({

success: false,

errors: errors.array()

});

}

//... (existing logic)

}

];

```

### Clean Code

1. \*\*Re-use Utility Functions\*\*

Functions like `getServiceItemById` and parts of other functions could be reusable to reduce code repetition.

```javascript

// This is good practice, and it can be slightly improved by making it reusable

exports.getServiceItemById = async (req, res, next) => {

try {

const serviceItem = await ServiceItem.findById(req.params.id);

if (!serviceItem) {

return res.status(404).json({ success: false, error: 'ServiceItem not found' });

}

req.serviceItem = serviceItem; // Save in request object for further use

next(); // Pass to the next middleware/controller function

} catch (error) {

console.error(`Error fetching service item by id: ${error.message}`, error);

res.status(500).json({

success: false,

error: error.message

});

}

};

```

2. \*\*DRY Principle in Controllers\*\*

If multiple endpoints require validating the existence of a `ServiceItem`, you can use the middleware created above.

```javascript

// Use the middleware

app.put('/service-items/:id', getServiceItemById, exports.updateServiceItem);

```

### Database

1. \*\*Handling `id` Not Found Errors\*\*

Ensure that `id` param validation that checks for valid MongoDB ObjectId is implemented.

```javascript

const mongoose = require('mongoose');

// Middleware to validate ObjectId

exports.validateObjectId = (req, res, next) => {

if (!mongoose.Types.ObjectId.isValid(req.params.id)) {

return res.status(400).json({ success: false, error: 'Invalid ID' });

}

next();

};

// Apply middleware in routes

app.put('/service-items/:id', validateObjectId, getServiceItemById, exports.updateServiceItem);

```

### Security

1. \*\*CSRF Protection\*\*

Properly protect routes that involve modifying data with CSRF tokens using middleware.

```javascript

const csrf = require('csurf');

const csrfProtection = csrf({ cookie: true });

app.post('/service-items', csrfProtection, exports.createServiceItem);

```

### Route Organization

1. \*\*Router Separation\*\*

Define routes in separate files using `express.Router()` for better code organization.

```javascript

// routes/serviceItemRoutes.js

const express = require('express');

const router = express.Router();

const serviceItemController = require('../controllers/serviceItemController');

router.post('/', serviceItemController.createServiceItem);

router.get('/', serviceItemController.getAllServiceItems);

router.get('/render', serviceItemController.renderServiceItemsPage);

router.get('/:id', serviceItemController.getServiceItemById, serviceItemController.getSingleServiceItem);

router.put('/:id', serviceItemController.validateObjectId, serviceItemController.getServiceItemById, serviceItemController.updateServiceItem);

router.delete('/:id', serviceItemController.validateObjectId, serviceItemController.getServiceItemById, serviceItemController.deleteServiceItem);

module.exports = router;

```

### Example Usage of Suggestions:

Combining these suggestions, here's how the `updateServiceItem` function can be refined:

```javascript

// Controller

exports.updateServiceItem = async (req, res) => {

if (!mongoose.Types.ObjectId.isValid(req.params.id)) {

return res.status(400).json({ success: false, error: 'Invalid ID' });

}

try {

const serviceItem = await ServiceItem.findByIdAndUpdate(req.params.id, req.body, {

new: true, runValidators: true

});

if (!serviceItem) {

console.log(`Service item not found with id: ${req.params.id}`);

return res.status(404).json({ success: false, error: 'ServiceItem not found' });

}

console.log(`Service item updated: ${serviceItem.name}`);

res.status(200).json({ success: true, data: serviceItem });

} catch (error) {

console.error(`Error updating service item: ${error.message}`, error);

res.status(400).json({ success: false, error: error.message });

}

};

// Route

router.put('/:id', validateObjectId, updateServiceItem);

```

### Final Thoughts

Incorporate these improvements to ensure the codebase is robust, maintainable, and secure. It will improve the overall quality of the application and facilitate easier future enhancements.

**authMiddleware**

Your `authMiddleware.js` file contains a simple middleware function to check if a user is authenticated. Overall, it looks good, but there are a few improvements and best practices that you can consider:

1. \*\*Consistency and Readability:\*\*

- Consider always using curly braces with `if` statements, even when they contain only one line. It improves readability and reduces the risk of introducing bugs when someone adds more code later.

2. \*\*Logging:\*\*

- Adding some logging can be helpful for debugging purposes, especially if you have to troubleshoot authentication issues.

3. \*\*Status Code:\*\*

- It might be useful to set the status code to `401 Unauthorized` or `403 Forbidden` before redirecting.

4. \*\*Code Comments:\*\*

- Adding comments can help others (and your future self) to understand the purpose of the middleware more easily.

5. \*\*Temporary Redirect:\*\*

- Using `res.status(302).redirect('/auth/login')` explicitly states that a temporary redirect is intended.

Here is your improved middleware considering the suggestions above:

```javascript

// middleware/authMiddleware.js

/\*\*

\* Middleware to verify if the user is authenticated.

\* @param {Object} req - Express request object

\* @param {Object} res - Express response object

\* @param {Function} next - Next middleware function

\*/

exports.isAuthenticated = (req, res, next) => {

// Check if session exists and if userId is present in the session

if (req.session && req.session.userId) {

return next();

} else {

console.log('Unauthorized access attempt, redirecting to login page.');

// Set status to 401 Unauthorized and redirect to login

res.status(401).redirect('/auth/login');

}

};

```

Some additional considerations depending on your application:

- \*\*Security Headers:\*\* Ensure your sessions are secure with appropriate cookie settings such as `httpOnly`, `secure`, and `sameSite`.

- \*\*Rate Limiting:\*\* Consider rate limiting for login endpoints to protect against brute force attacks.

- \*\*Session Management:\*\* Validate and handle expired or invalid sessions correctly to enhance security.

These improvements should enhance the robustness and maintainability of your authentication middleware.

**getClient.js**

Your code is well-structured and seems to achieve its intended purpose. However, I have a few suggestions for improvements and best practices:

### Error Handling

1. \*\*Verbose Logging:\*\* Your error logging is good, but sometimes it's helpful to make it even more verbose depending on the situation, such as logging the stack trace for deeper insights.

### Code Readability

2. \*\*Consistent Response Structure:\*\* Ensure your server responses follow a consistent structure for easier front-end handling. For example, you could always include a status field in your JSON responses.

3. \*\*Validation:\*\* It's a good idea to validate the `req.params.id` to check if it's a valid ID format before passing it to the database query.

### Performance

4. \*\*Lean Queries:\*\* If the entire client object is not required, you can optimize the database call to return only necessary fields.

### Security

5. \*\*Sanitization:\*\* Always sanitize the inputs, especially if you pass them into database queries. This helps to prevent security vulnerabilities like NoSQL injection.

### Refactored Code

Here's a slightly refactored version incorporating the suggestions above:

```javascript

const Client = require('../models/clientModel');

const mongoose = require('mongoose');

const getClient = async (req, res, next) => {

const clientId = req.params.id;

// Validate ObjectId format if using MongoDB

if (!mongoose.Types.ObjectId.isValid(clientId)) {

return res.status(400).json({ status: "error", message: 'Invalid client ID format' });

}

try {

// Only fetch the needed fields

const client = await Client.findById(clientId).select('name email'); // adjusting fields as needed

if (!client) {

return res.status(404).json({ status: "error", message: 'Client not found' });

}

req.client = client; // Attach client to the request object

next();

} catch (error) {

console.error(`Error fetching client with ID ${clientId}: ${error.message}`, error);

res.status(500).json({ status: "error", message: 'Failed to fetch client', error: error.message });

}

};

module.exports = getClient;

```

### Changes Explained:

1. \*\*Validation:\*\* Added a check to validate the client ID format before querying the database.

2. \*\*Lean Queries:\*\* Used `.select('name email')` to select only specific fields. Adjust this to fit which fields are necessary.

3. \*\*Consistent Response Structure:\*\* Added a 'status' field in JSON responses.

These improvements can help make your code more robust, readable, and efficient.

**clientModel**

The provided code defines a Mongoose model for a `Client` schema, includes some middleware hooks for logging, and exports the model. Here's a detailed analysis:

### Code Analysis

1. \*\*Schema Definition\*\*

- The schema is defined correctly with various fields like `clientId`, `companyName`, `name`, etc.

- Validation constraints such as `required` and `unique` are appropriately applied.

2. \*\*Middleware Hooks\*\*

- \*\*Pre-save hook\*\*: Logs whether a client is being created or updated.

- \*\*Post-save hook\*\*: Logs a message after a client has been saved.

- \*\*Post-remove hook\*\*: Logs a message after a client has been removed.

### Possible Improvements and Suggestions

1. \*\*Validation and Normalization\*\*

- \*\*Email Validation\*\*: Ensure emails are in a proper format.

```javascript

email: { type: String, required: true, unique: true, match: /.+\@.+\..+/ }

```

- \*\*Optional Fields Default Value\*\*: Consider using default values for optional fields. This can prevent `undefined` values.

```javascript

companyName: { type: String, default: '' }

```

2. \*\*Consistency Logging\*\*

- Enhance log messages to provide more context, such as including `clientId` for better traceability.

```javascript

clientSchema.pre('save', function(next) {

if (!this.isNew) {

console.log(`Updating existing client with ID: ${this.clientId}`);

} else {

console.log(`Creating a new client with ID: ${this.clientId}`);

}

next();

});

clientSchema.post('save', function(doc, next) {

console.log(`Client ${doc.name} (ID: ${doc.clientId}) saved to database`);

next();

});

clientSchema.post('remove', function(doc, next) {

console.log(`Client ${doc.name} (ID: ${doc.clientId}) removed from database`);

next();

});

```

3. \*\*Schema Indexes\*\*

- For fields that are often queried, consider creating indexes to improve query performance.

```javascript

clientSchema.index({ email: 1 });

clientSchema.index({ clientId: 1 });

```

4. \*\*Error Handling\*\*

- Log errors in middleware hooks for better traceability.

```javascript

clientSchema.post('save', function(error, doc, next) {

if (error) {

console.error(`Error saving client ${doc.name}: ${error}`);

}

next(error);

});

clientSchema.post('remove', function(error, doc, next) {

if (error) {

console.error(`Error removing client ${doc.name}: ${error}`);

}

next(error);

});

```

5. \*\*Optional Schema Methods or Statics\*\*

- Consider adding instance methods or static methods to the schema for commonly used functions.

```javascript

clientSchema.methods.findSimilarClients = function(cb) {

return mongoose.model('Client').find({ state: this.state }, cb);

};

```

### Final Version with Improvements

```javascript

const mongoose = require('mongoose');

const clientSchema = new mongoose.Schema({

clientId: { type: String, required: true, unique: true },

companyName: { type: String, default: '' },

name: { type: String, required: true },

email: { type: String, required: true, unique: true, match: /.+\@.+\..+/ },

phoneNumber: { type: String, default: '' },

streetAddress: { type: String, default: '' },

city: { type: String, default: '' },

state: { type: String, default: '' },

zip: { type: String, default: '' }

});

clientSchema.index({ email: 1 });

clientSchema.index({ clientId: 1 });

clientSchema.pre('save', function(next) {

if (!this.isNew) {

console.log(`Updating existing client with ID: ${this.clientId}`);

} else {

console.log(`Creating a new client with ID: ${this.clientId}`);

}

next();

});

clientSchema.post('save', function(doc, next) {

console.log(`Client ${doc.name} (ID: ${doc.clientId}) saved to database`);

next();

});

clientSchema.post('remove', function(doc, next) {

console.log(`Client ${doc.name} (ID: ${doc.clientId}) removed from database`);

next();

});

clientSchema.post('save', function(error, doc, next) {

if (error) {

console.error(`Error saving client ${doc.name}: ${error}`);

}

next(error);

});

clientSchema.post('remove', function(error, doc, next) {

if (error) {

console.error(`Error removing client ${doc.name}: ${error}`);

}

next(error);

});

const Client = mongoose.model('Client', clientSchema);

module.exports = Client;

```

These suggestions enhance code readability, maintainability, and ensure robust error handling and validation.

**quoteModel.js**

Your code for the `quoteModel.js` file is well-structured and employs Mongoose schemas effectively. However, there are several areas where improvements can be made for better functionality, clarity, and error handling. Here's an analysis along with suggestions for improvements:

### Analysis

1. \*\*Item Schema\*\*:

- You defined `itemSchema`, but it seems unused. Clarify its purpose or remove it if it's unnecessary.

2. \*\*Field Types\*\*:

- Consider using `Number` types for `price` fields with `min: 0` for validation to ensure only non-negative values.

- For `totalPrice`, it’s inconsistent with `total`. Either keep it or remove redundancy.

3. \*\*Quote Schema\*\*:

- It's well-structured but has some redundant fields, such as `totalPrice` (you already have `total`).

- `mixed` type (`priceBreakdown: mongoose.Schema.Types.Mixed`) can store any JavaScript data type, ensure it doesn't cause inconsistency. Consider using a more rigid schema if possible.

- `frequency` should include more options or handle cases where intervals are needed (like every three weeks).

4. \*\*Pre/Post Save Middleware\*\*:

- Good for logging but consider adding more business logic (e.g., calculating `total`).

5. \*\*Default Values\*\*:

- Default value for schema methods like `quoteId: { type: String, default: uuidv4 }` should be `default: () => uuidv4()` for Mongoose constructor parameter consistency.

### Improvements and Suggestions

1. \*\*Field Type Validation and Defaults\*\*:

- Update schema definitions with strict type validations and sensible defaults to avoid errors.

```javascript

const itemSchema = new mongoose.Schema({

description: { type: String, required: true },

quantity: { type: Number, required: true, min: 0 },

price: { type: Number, required: true, min: 0 },

isService: { type: Boolean, required: true },

}, { \_id: false });

```

2. \*\*More Robust Middleware\*\*:

- Add business logic, e.g., computing totals before saving.

```javascript

quoteSchema.pre('save', function (next) {

this.total = this.subtotal + (this.subtotal \* (this.taxRate / 100));

console.log(`Quote ID: ${this.quoteId} Total calculated as: ${this.total}`);

next();

});

```

3. \*\*Better Default and Enum Handling\*\*:

- Ensure robust use of enums to prevent invalid states.

- Provide robust schemas for documents with nested data.

```javascript

const quoteSchema = new mongoose.Schema({

...

total: { type: Number, required: true, default: 0 },

total: { type: Number, default: function() { return this.subtotal \* (1 + this.taxRate / 100); } }, // Dynamic default total computation

...

});

```

4. \*\*Better Error Handling And Additional Validations\*\*:

- Handling Mongoose’s built-in validations or custom validations to improve data integrity.

### Final Revised Code:

```javascript

const mongoose = require('mongoose');

const { v4: uuidv4 } = require('uuid');

const itemSchema = new mongoose.Schema({

description: { type: String, required: true },

quantity: { type: Number, required: true, min: 0 },

price: { type: Number, required: true, min: 0 },

isService: { type: Boolean, required: true },

}, { \_id: false });

const serviceItemSchema = new mongoose.Schema({

serviceItemId: { type: mongoose.Schema.Types.ObjectId, ref: 'ServiceItem', required: true },

quantity: { type: Number, required: true, min: 0 },

customPrice: Number,

}, { \_id: false });

const attachmentSchema = new mongoose.Schema({

savedFilename: { type: String, required: true },

originalFilename: { type: String, required: true }

}, { \_id: false });

const quoteSchema = new mongoose.Schema({

quoteId: { type: String, default: () => uuidv4() },

clientName: { type: String, required: true },

clientId: { type: mongoose.Schema.Types.ObjectId, ref: 'Client', required: true },

title: String,

scopeOfWork: String,

status: {

type: String,

enum: ['sent', 'opened', 'clicked'],

default: 'sent'

},

serviceType: {

type: String,

required: true,

enum: ['Recurring', 'One-Time Deep Clean', 'Move In/Move Out']

},

frequency: {

type: String,

enum: ['Weekly', 'Bi-Weekly', 'Monthly'],

default: null

},

initialCleaningOptions: [{

type: String,

enum: ['Walls', 'Windows', 'Inside Fridge', 'Inside Stove', 'Underneath Furniture', 'Other']

}],

serviceItems: [serviceItemSchema],

attachments: [attachmentSchema],

contracts: [attachmentSchema],

subtotal: { type: Number, default: 0 },

taxRate: { type: Number, default: 7.5 },

total: { type: Number, default: function() { return this.subtotal \* (1 + this.taxRate / 100); } },

totalPrice: Number,

priceBreakdown: mongoose.Schema.Types.Mixed

});

quoteSchema.pre('save', function (next) {

this.total = this.subtotal + (this.subtotal \* (this.taxRate / 100));

console.log(`Quote ID: ${this.quoteId} Total: ${this.total}`);

next();

});

quoteSchema.post('save', async function(doc, next) {

console.log(`Quote ${doc.quoteId} saved to database`);

next();

});

const Quote = mongoose.model('Quote', quoteSchema);

module.exports = Quote;

```

### Improved Summary:

1. Added robust validation to schema definitions.

2. Enhanced default value handling and total computation logic.

3. Kept the middleware for logging and essential updates.

4. Suggested removing `totalPrice` as it’s redundant.

5. Emphasized better practice for dynamic default values using Mongoose functions.

These changes improve data integrity, code clarity, and functionality in your Mongoose schemas and models.

**serviceItem.js**

The provided code defines a Mongoose schema and model for a `ServiceItem`. It includes functionality such as pre and post hooks for saving and removing items, along with schema validation and indexing for efficient querying. Below is an analysis of the code, highlighting any errors and suggesting improvements:

### Analysis

1. \*\*Schema Definition\*\*

- The schema defines the fields (`name`, `description`, `category`, `units`, and `price`) correctly with validation rules.

- `description` and `category` fields are not required, but `category` includes an enum for validation which is useful.

- `units` field has an enum for validation, ensuring only specified values are allowed.

2. \*\*Indexing\*\*

- An index is created on `name` and `category` with `unique: true`. This enforces the uniqueness constraint on these fields in combination, which can be useful in avoiding duplicate entries.

3. \*\*Pre and Post Hooks\*\*

- `pre('save')`: This correctly logs a message before saving the document.

- `post('save')`: This correctly logs a message after the document has been saved.

- `post('remove')`: This correctly logs a message after the document has been removed from the database.

### Improvements and Suggestions

1. \*\*Error Handling in Hooks\*\*

- While the functions currently just log messages, it would be good practice to handle potential errors as well. For example, making sure `next` is called even if an error occurs.

```javascript

serviceItemSchema.pre('save', async function(next) {

try {

console.log(`Saving item: ${this.name}`);

next();

} catch (error) {

next(error);

}

});

serviceItemSchema.post('save', function(doc, next) {

try {

console.log(`Item ${doc.name} saved to database`);

next();

} catch (error) {

next(error);

}

});

serviceItemSchema.post('remove', function(doc, next) {

try {

console.log(`Item ${doc.name} removed from database`);

next();

} catch (error) {

next(error);

}

});

```

2. \*\*Schema Validation Messages\*\*

- Adding custom error messages can improve the clarity of validation errors.

```javascript

const serviceItemSchema = new mongoose.Schema({

name: { type: String, required: [true, 'Name is required'] },

description: { type: String, required: false },

category: { type: String, required: false, enum: { values: ['Standard', 'Extras', 'Package', 'Service'], message: '{VALUE} is not a valid category' } },

units: { type: String, required: [true, 'Units are required'], enum: { values: ['Per Item', 'Per Hour'], message: '{VALUE} is not a valid unit' } },

price: { type: Number, required: [true, 'Price is required'] },

});

```

3. \*\*Schema Options\*\*

- Adding timestamps to the schema can help track the creation and update times of documents.

```javascript

const serviceItemSchema = new mongoose.Schema({

name: { type: String, required: true },

description: { type: String, required: false },

category: { type: String, required: false, enum: ['Standard', 'Extras', 'Package', 'Service'] },

units: { type: String, required: true, enum: ['Per Item', 'Per Hour'] },

price: { type: Number, required: true },

},

{

timestamps: true

});

```

4. \*\*Code Comments\*\*

- While the code already has comments, adding more detailed comments, especially about the purpose of each hook, could further improve maintainability.

### Final Code with Improvements

```javascript

const mongoose = require('mongoose');

const serviceItemSchema = new mongoose.Schema({

name: { type: String, required: [true, 'Name is required'] },

description: { type: String, required: false },

category: { type: String, required: false, enum: { values: ['Standard', 'Extras', 'Package', 'Service'], message: '{VALUE} is not a valid category' } },

units: { type: String, required: [true, 'Units are required'], enum: { values: ['Per Item', 'Per Hour'], message: '{VALUE} is not a valid unit' } },

price: { type: Number, required: [true, 'Price is required'] },

}, {

timestamps: true

});

// Creating indexes for 'name' and 'category' for efficient querying

serviceItemSchema.index({ name: 1, category: 1 }, { unique: true });

serviceItemSchema.pre('save', async function(next) {

try {

console.log(`Saving item: ${this.name}`);

next();

} catch (error) {

next(error);

}

});

serviceItemSchema.post('save', function(doc, next) {

try {

console.log(`Item ${doc.name} saved to database`);

next();

} catch (error) {

next(error);

}

});

serviceItemSchema.post('remove', function(doc, next) {

try {

console.log(`Item ${doc.name} removed from database`);

next();

} catch (error) {

next(error);

}

});

const ServiceItem = mongoose.model('ServiceItem', serviceItemSchema);

module.exports = ServiceItem;

```

Overall, the original code is functional, but following these suggestions can enhance its robustness, maintainability, and clarity.

**Users.js**

Your JavaScript (Express.js) code defining the User model with Mongoose and bcrypt for hashing passwords looks fairly well-structured. However, there are always potential improvements and suggestions for better practices and potential error handling. Here is a detailed analysis:

### Errors and Improvements:

1. \*\*Email Validation:\*\*

- Currently, the email field doesn't have any validation. It's a good idea to add an email validation regex pattern to ensure the email format is correct.

```js

email: {

type: String,

match: [/\S+@\S+\.\S+/, 'is invalid'],

required: true

},

```

2. \*\*Phone Number Validation:\*\*

- Similarly, validating the phone number format can be useful. This can be done using a regex or using a library.

```js

phoneNumber: {

type: String,

match: [/^\d{10}$/, 'is invalid']

},

```

3. \*\*Refactor `hoursOfOperation` Object:\*\*

- The `hoursOfOperation` object could be defined slightly more elegantly and with more granularity, like this:

```js

hoursOfOperation: {

monday: { type: Map, of: String },

tuesday: { type: Map, of: String },

wednesday: { type: Map, of: String },

thursday: { type: Map, of: String },

friday: { type: Map, of: String },

saturday: { type: Map, of: String },

sunday: { type: Map, of: String }

},

```

4. \*\*Unique Indexes for Important Fields:\*\*

- It's good to ensure that `email`, in addition to `username`, is unique if your application expects unique emails.

```js

email: {

type: String,

match: [/\S+@\S+\.\S+/, 'is invalid'],

required: true,

unique: true

},

```

5. \*\*Timestamps:\*\*

- Adding timestamps to the schema (`createdAt`, `updatedAt`) can be very useful for record-keeping.

```js

const userSchema = new mongoose.Schema({

// fields...

}, {

timestamps: true

});

```

6. \*\*Error Handling Improvement:\*\*

- The current error handling is good, but adding specific error messages might help in debugging.

### Suggestions:

1. \*\*Mongoose Validation:\*\*

- Make use of Mongoose's built-in validation for fields such as `username` and `password`. E.g., setting `minlength` for the password.

```js

password: {

type: String,

required: true,

minlength: [8, 'Password too short']

},

```

2. \*\*Updating Password Hashing:\*\*

- There's no need to hash the password when it's not modified. Add a better condition for it.

```js

userSchema.pre('save', async function(next) {

if (!this.isModified('password')) return next(); // Move return to the same line

//...

});

```

3. \*\*Async/Await in comparePassword Method:\*\*

- Consider using `async`/`await` for consistency with modern JavaScript practices.

```js

userSchema.methods.comparePassword = async function(candidatePassword) {

try {

return await bcrypt.compare(candidatePassword, this.password);

} catch (err) {

throw new Error(err);

}

};

```

### Final Revised Code:

Below is your code with above adjustments:

```js

const mongoose = require('mongoose');

const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({

username: { type: String, required: true, unique: true },

password: { type: String, required: true, minlength: [8, 'Password too short'] },

businessName: { type: String },

businessLogo: { type: String },

name: { type: String },

phoneNumber: { type: String, match: [/^\d{10}$/, 'is invalid'] },

email: { type: String, match: [/\S+@\S+\.\S+/, 'is invalid'], required: true, unique: true },

streetAddress: { type: String },

city: { type: String },

state: { type: String },

zipCode: { type: String },

hoursOfOperation: {

monday: { type: Map, of: String },

tuesday: { type: Map, of: String },

wednesday: { type: Map, of: String },

thursday: { type: Map, of: String },

friday: { type: Map, of: String },

saturday: { type: Map, of: String },

sunday: { type: Map, of: String }

},

websiteURL: { type: String },

socialMediaLinks: { type: String }

}, {

timestamps: true

});

userSchema.pre('save', async function(next) {

if (!this.isModified('password')) return next();

try {

const salt = await bcrypt.genSalt(10);

this.password = await bcrypt.hash(this.password, salt);

next();

} catch (error) {

next(error);

}

});

userSchema.methods.comparePassword = async function(candidatePassword) {

try {

return await bcrypt.compare(candidatePassword, this.password);

} catch (err) {

throw new Error(err);

}

};

const User = mongoose.model('User', userSchema);

module.exports = User;

```

These changes should make your code more robust, readable, and maintainable.

**ajaxCsrfTokenHandler.js**

The provided code snippet shows a JavaScript function `logToServer` designed to send log messages to a server endpoint via an HTTP POST request. Here is an in-depth analysis, including errors, improvements, and suggestions:

### Analysis

#### Function Purpose

This function sends log messages of different levels (like info, warning, error) to a server endpoint (`/logs`) using the Fetch API.

### Improvements and Suggestions

1. \*\*Check Response Status:\*\*

You should handle the response to ensure the logging was successful. Just catching the fetch error isn't sufficient. You need to verify if the server actually received and processed the log message.

```javascript

function logToServer(level, message) {

fetch('/logs', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ level, message })

})

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

})

.then(data => {

console.debug('Log sent successfully:', data);

})

.catch(error => console.error('Error logging to server:', error));

}

```

2. \*\*Handle Network Errors Gracefully:\*\*

Instead of just logging the error to the console, consider retrying the request or queuing it for later if the network error persists.

3. \*\*Validate Inputs:\*\*

Ensure that the `level` and `message` are of the expected types before attempting to send them.

```javascript

function logToServer(level, message) {

if (typeof level !== 'string' || typeof message !== 'string') {

console.error('Invalid log level or message');

return;

}

fetch('/logs', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ level, message })

})

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

})

.then(data => {

console.debug('Log sent successfully:', data);

})

.catch(error => console.error('Error logging to server:', error));

}

```

4. \*\*Security Concerns:\*\*

Be cautious about the information you log. Ensure sensitive information is not being logged inadvertently.

5. \*\*Modularity:\*\*

Consider making `logToServer` part of a larger logging library or class to better manage different log levels and configurations.

```javascript

class ClientLogger {

constructor(endpoint) {

this.endpoint = endpoint;

}

log(level, message) {

if (typeof level !== 'string' || typeof message !== 'string') {

console.error('Invalid log level or message');

return;

}

fetch(this.endpoint, {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ level, message })

})

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

})

.then(data => {

console.debug('Log sent successfully:', data);

})

.catch(error => console.error('Error logging to server:', error));

}

}

// Usage:

const logger = new ClientLogger('/logs');

logger.log('info', 'This is an informational message');

```

### Summary

While the initial implementation of `logToServer` does its job minimally, adding more error handling, input validation, security considerations, and modularity can significantly improve it. Adopting these improvements can result in more robust and maintainable code.

**clientSearch.js**

Your JavaScript code appears to be well-structured and functional for the most part. However, there are several improvements and considerations you might want to address:

1. \*\*DRY (Don't Repeat Yourself) Principle\*\*:

- There's repeated code for populating client preview fields (`clientNamePreview`, `clientEmailPreview`, etc.). Consider creating a function to handle this.

2. \*\*Error Handling\*\*:

- Error handling could be improved by providing more specific messages to the user instead of just logging to the console.

3. \*\*Input Validation\*\*:

- Ensure all user inputs are properly validated and sanitized to prevent issues like XSS attacks.

4. \*\*CSRF Token\*\*:

- It's good that you're using CSRF tokens, but ensure that these tokens are securely handled and stored.

5. \*\*Event Delegation\*\*:

- Consider using event delegation for dynamically created elements, like the client list items, to improve performance and manageability.

6. \*\*Code Readability\*\*:

- Add more comments to explain the purpose of different sections, which will make the code easier to maintain.

7. \*\*Defensive Programming\*\*:

- Always check that the required elements exist before attempting to access or manipulate them.

Here’s the refactored code with suggested improvements:

```javascript

document.addEventListener('DOMContentLoaded', function() {

const clientSearchInput = document.getElementById('clientSearch');

const clientResults = document.getElementById('clientResults');

const clientIdInput = document.getElementById('clientId');

const clientNameInput = document.getElementById('clientName');

const clientPreview = document.getElementById('clientPreview');

const clientNamePreview = document.getElementById('clientNamePreview');

const clientEmailPreview = document.getElementById('clientEmailPreview');

const clientPhonePreview = document.getElementById('clientPhonePreview');

const clientAddressPreview = document.getElementById('clientAddressPreview');

const removeClientBtn = document.getElementById('removeClientBtn');

const editClientBtn = document.getElementById('editClientBtn');

const createClientForm = document.getElementById('createClientForm');

const editClientForm = document.getElementById('editClientForm');

function populateClientPreview(client) {

clientNamePreview.textContent = client.name;

clientEmailPreview.textContent = client.email;

clientPhonePreview.textContent = client.phoneNumber;

clientAddressPreview.textContent = `${client.streetAddress}, ${client.city}, ${client.state}, ${client.zip}`;

clientPreview.classList.remove('d-none');

}

async function fetchClients(query) {

try {

const response = await fetch(`/clients/search?query=${query}`, { cache: 'no-store' });

if (!response.ok) throw new Error('Network response was not ok');

return await response.json();

} catch (error) {

console.error('Error fetching clients:', error);

clientResults.innerHTML = '<div class="list-group-item">Error fetching clients. Please try again later.</div>';

return [];

}

}

async function handleClientSearchInput(event) {

const query = event.target.value.trim();

clientResults.innerHTML = '';

if (query.length > 2) {

const clients = await fetchClients(query);

if (clients.length > 0) {

clients.forEach(client => {

const clientItem = document.createElement('a');

clientItem.href = '#';

clientItem.className = 'list-group-item list-group-item-action';

clientItem.textContent = client.name;

clientItem.addEventListener('click', () => handleClientItemClick(client));

clientResults.appendChild(clientItem);

});

} else {

displayNoResults();

}

}

}

function handleClientItemClick(client) {

clientSearchInput.value = client.name;

clientIdInput.value = client.\_id;

clientNameInput.value = client.name;

populateClientPreview(client);

clientResults.innerHTML = '';

}

function displayNoResults() {

const noResults = document.createElement('div');

noResults.className = 'list-group-item';

noResults.textContent = 'No clients found. Would you like to ';

const createLink = document.createElement('a');

createLink.href = '#';

createLink.textContent = 'create a new client?';

createLink.addEventListener('click', () => {

const createClientModal = new bootstrap.Modal(document.getElementById('createClientModal'));

createClientModal.show();

});

noResults.appendChild(createLink);

clientResults.appendChild(noResults);

}

async function submitClientForm(form, url, method) {

const formData = new FormData(form);

const data = {};

formData.forEach((value, key) => {

data[key] = value;

});

try {

const response = await fetch(url, {

method: method,

headers: {

'Content-Type': 'application/json',

'CSRF-Token': data.\_csrf

},

body: JSON.stringify(data)

});

if (!response.ok) throw new Error('Network response was not ok');

return await response.json();

} catch (error) {

console.error(`Error ${method === 'POST' ? 'creating' : 'updating'} client:`, error);

alert(`Error ${method === 'POST' ? 'creating' : 'updating'} client. Please try again.`);

return null;

}

}

async function handleClientFormSubmit(form, event, isEdit) {

event.preventDefault();

const method = isEdit ? 'PUT' : 'POST';

const url = isEdit ? `/clients/${form.clientId.value}` : '/clients';

const result = await submitClientForm(form, url, method);

if (result && result.success) {

clientSearchInput.value = result.data.name;

clientIdInput.value = result.data.\_id;

clientNameInput.value = result.data.name;

populateClientPreview(result.data);

clientResults.innerHTML = '';

const modal = bootstrap.Modal.getInstance(isEdit ? document.getElementById('editClientModal') : document.getElementById('createClientModal'));

modal.hide();

}

}

async function populateEditClientForm(clientId) {

try {

const response = await fetch(`/clients/${clientId}`, { cache: 'no-store' });

if (!response.ok) throw new Error('Network response was not ok');

const client = await response.json();

document.getElementById('editClientId').value = client.\_id;

document.getElementById('editClientName').value = client.name;

document.getElementById('editClientEmail').value = client.email;

document.getElementById('editClientPhoneNumber').value = client.phoneNumber;

document.getElementById('editClientStreetAddress').value = client.streetAddress;

document.getElementById('editClientCity').value = client.city;

document.getElementById('editClientState').value = client.state;

document.getElementById('editClientZip').value = client.zip;

const editClientModal = new bootstrap.Modal(document.getElementById('editClientModal'));

editClientModal.show();

} catch (error) {

console.error('Error fetching client for edit:', error);

alert('Error fetching client data. Please try again.');

}

}

function handleRemoveClient(event) {

event.preventDefault();

if (confirm('Are you sure you want to remove this client from the quote?')) {

clientIdInput.value = '';

clientNameInput.value = '';

clientSearchInput.value = '';

clientNamePreview.textContent = '';

clientEmailPreview.textContent = '';

clientPhonePreview.textContent = '';

clientAddressPreview.textContent = '';

clientPreview.classList.add('d-none');

}

}

// Event Listeners

if (clientSearchInput) clientSearchInput.addEventListener('input', handleClientSearchInput);

if (removeClientBtn) removeClientBtn.addEventListener('click', handleRemoveClient);

if (editClientBtn) editClientBtn.addEventListener('click', event => {

event.preventDefault();

const clientId = clientIdInput.value;

if (clientId) populateEditClientForm(clientId);

});

if (createClientForm) createClientForm.addEventListener('submit', event => handleClientFormSubmit(createClientForm, event, false));

if (editClientForm) editClientForm.addEventListener('submit', event => handleClientFormSubmit(editClientForm, event, true));

});

```

### Changes Made:

1. Extracted repeated logic into helper functions (`populateClientPreview`, `fetchClients`, `submitClientForm`).

2. Added error handling improvements.

3. Used a single event handler setup for `submit` events on both forms (`handleClientFormSubmit`), which is parameterized to handle both create and edit actions.

4. Improved readability by adding more comments where necessary.

5. Used more descriptive function names for clarity.

These improvements aim to make the script more maintainable, readable, and user-friendly.

**csrfTokenRefresher.js**

Your code looks pretty solid, but there are several improvements and considerations you could incorporate:

### Improvements:

1. \*\*Error Handling\*\*:

Add error handling to manage any potential issues with the fetch request. This will help you gracefully handle any network errors or issues with the server response.

2. \*\*Optimized Function Naming\*\*:

The functions are well-named; however, adding JSDoc comments can improve readability and maintainability, helping other developers understand the purpose and usage of these functions quickly.

3. \*\*CSRF Token Propagation\*\*:

If the CSRF token is used in subsequent API calls, it should be stored in a global variable or a state management solution (e.g., Redux for a React app).

4. \*\*Avoid Inline Await\*\*:

Inline use of `await` inside a loop (like a `.forEach()` loop) can lead to unexpected behavior since `forEach` does not inherently support asynchronous operation. For your current use case, this is not an issue as you're not requiring async operations inside the loop, but this is a point often worth noting.

### Example of Improved Code:

```javascript

/\*\*

\* Update CSRF token inputs on the page

\* @param {string} newToken - The new CSRF token

\*/

const updateCsrfToken = (newToken) => {

const csrfInputs = document.querySelectorAll('input[name="\_csrf"]');

csrfInputs.forEach(input => {

input.value = newToken;

});

};

/\*\*

\* Fetch data from the given URL and update CSRF token if provided

\* @param {string} url - The endpoint URL

\* @returns {Promise<Object>} The JSON response data

\* @throws Will throw an error if the fetch operation fails

\*/

const fetchDataWithCsrfUpdate = async (url) => {

try {

const response = await fetch(url, { credentials: 'same-origin' });

if (!response.ok) {

throw new Error(`HTTP error! status: ${response.status}`);

}

const data = await response.json();

if (data.csrfToken) {

updateCsrfToken(data.csrfToken);

}

return data;

} catch (error) {

console.error('Error fetching data:', error);

throw error;

}

};

// Example usage

fetchDataWithCsrfUpdate('/some-endpoint')

.then(data => {

console.log('Data fetched with CSRF token update:', data);

})

.catch(error => {

console.error('Error during fetch operation:', error);

});

```

### Key Considerations:

1. \*\*Error Handling\*\*:

The improved code now catches errors during the fetch operation and logs them, providing better traceability.

2. \*\*HTTP Status Check\*\*:

The `!response.ok` check ensures that non-2xx HTTP responses are properly handled.

3. \*\*JSDoc Comments\*\*:

These comments improve readability and provide clear guidance on the purpose and parameters of the functions.

4. \*\*`credentials: 'same-origin'`\*\*:

This is correctly used to ensure that cookies are sent with cross-origin requests, which is important for CSRF protection.

These improvements make the code more robust and maintainable.

**dynamicServiceItemPopulation.js**

Your JavaScript (working within an Express.js or EJS template) code appears to be generally well-structured and functional. However, there are a few areas where you could improve it for maintainability, readability, and reliability. Here are some observations, potential improvements, and suggestions:

1. \*\*Code Organization & Readability\*\*: Consider grouping related functional blocks and adding comments to make the code easier to understand and maintain.

2. \*\*Error Handling\*\*: While you are using try-catch blocks, it would be beneficial to provide more specific error messages or handling mechanisms.

3. \*\*Repeated Code\*\*: You are updating the client details in multiple places. This repetitive code could be extracted into a function to avoid redundancy and improve maintainability.

4. \*\*Null Checks\*\*: While you are doing null checks for certain elements, it's advisable to ensure all DOM dependencies are checked to avoid runtime errors.

5. \*\*HTML Escaping\*\*: Ensure that the client's input is sanitized to prevent XSS (Cross-Site Scripting) attacks.

6. \*\*Modern JavaScript Practices\*\*: Make use of modern JavaScript features like `const` and `let` rather than `var`, and possibly consider using `async/await` more consistently.

Here’s a cleaner version of your script with these suggestions applied:

```javascript

document.addEventListener('DOMContentLoaded', function() {

// DOM elements

const elements = {

clientSearchInput: document.getElementById('clientSearch'),

clientResults: document.getElementById('clientResults'),

clientIdInput: document.getElementById('clientId'),

clientNameInput: document.getElementById('clientName'),

clientEmailInput: document.getElementById('clientEmail'),

clientPhoneInput: document.getElementById('clientPhone'),

clientAddressInput: document.getElementById('clientAddress'),

clientPreview: document.getElementById('clientPreview'),

clientNamePreview: document.getElementById('clientNamePreview'),

clientEmailPreview: document.getElementById('clientEmailPreview'),

clientPhonePreview: document.getElementById('clientPhonePreview'),

clientAddressPreview: document.getElementById('clientAddressPreview'),

removeClientBtn: document.getElementById('removeClientBtn'),

editClientBtn: document.getElementById('editClientBtn'),

serviceTypeSelect: document.getElementById('serviceType'),

frequencyDiv: document.getElementById('frequencyDiv'),

initialCleaningDiv: document.getElementById('initialCleaningDiv'),

createClientForm: document.getElementById('createClientForm'),

editClientForm: document.getElementById('editClientForm')

};

// Function to toggle visibility of additional options based on service type

function toggleServiceOptions() {

const serviceType = elements.serviceTypeSelect.value;

const isRecurring = serviceType === 'Recurring';

const isSpecial = serviceType === 'One-Time Deep Clean' || serviceType === 'Move In/Move Out';

elements.frequencyDiv.classList.toggle('d-none', !isRecurring);

elements.initialCleaningDiv.classList.toggle('d-none', !isSpecial);

}

// Function to hide the dropdown

function hideDropdown() {

elements.clientResults.innerHTML = '';

}

// Function to update client details in input fields and previews

function updateClientDetails(client) {

if (elements.clientIdInput) elements.clientIdInput.value = client.\_id;

if (elements.clientNameInput) elements.clientNameInput.value = client.name;

if (elements.clientEmailInput) elements.clientEmailInput.value = client.email;

if (elements.clientPhoneInput) elements.clientPhoneInput.value = client.phoneNumber;

if (elements.clientAddressInput) elements.clientAddressInput.value = `${client.streetAddress}, ${client.city}, ${client.state}, ${client.zip}`;

if (elements.clientNamePreview) elements.clientNamePreview.textContent = client.name;

if (elements.clientEmailPreview) elements.clientEmailPreview.textContent = client.email;

if (elements.clientPhonePreview) elements.clientPhonePreview.textContent = client.phoneNumber;

if (elements.clientAddressPreview) elements.clientAddressPreview.textContent = `${client.streetAddress}, ${client.city}, ${client.state}, ${client.zip}`;

if (elements.clientPreview) elements.clientPreview.classList.remove('d-none');

}

// Add event listener for the client search input

if (elements.clientSearchInput) {

elements.clientSearchInput.addEventListener('input', async function() {

const query = elements.clientSearchInput.value.trim();

if (query.length > 2) {

try {

const response = await fetch(`/clients/search?query=${query}`, { cache: 'no-store' });

if (!response.ok) throw new Error('Network response was not ok');

const clients = await response.json();

elements.clientResults.innerHTML = '';

if (clients.length > 0) {

clients.forEach(client => {

const clientItem = document.createElement('a');

clientItem.href = '#';

clientItem.className = 'list-group-item list-group-item-action';

clientItem.textContent = client.name;

clientItem.addEventListener('click', function(event) {

event.preventDefault();

elements.clientSearchInput.value = client.name;

updateClientDetails(client);

hideDropdown();

});

elements.clientResults.appendChild(clientItem);

});

} else {

const noResults = document.createElement('div');

noResults.className = 'list-group-item';

noResults.textContent = 'No clients found. Would you like to ';

const createLink = document.createElement('a');

createLink.href = '#';

createLink.textContent = 'create a new client?';

createLink.addEventListener('click', function() {

const createClientModal = new bootstrap.Modal(document.getElementById('createClientModal'));

createClientModal.show();

});

noResults.appendChild(createLink);

elements.clientResults.appendChild(noResults);

}

} catch (error) {

console.error('Error fetching clients:', error);

elements.clientResults.innerHTML = '<div class="list-group-item">Error fetching clients. Please try again later.</div>';

}

} else {

elements.clientResults.innerHTML = '';

}

});

}

// Event listeners for creating, removing, and editing clients

if (elements.createClientForm) {

elements.createClientForm.addEventListener('submit', async function(event) {

event.preventDefault();

const formData = new FormData(elements.createClientForm);

const data = {};

formData.forEach((value, key) => data[key] = value);

try {

const response = await fetch('/clients', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'CSRF-Token': data.\_csrf

},

body: JSON.stringify(data)

});

if (!response.ok) throw new Error('Network response was not ok');

const newClient = await response.json();

if (newClient.success) {

elements.clientSearchInput.value = newClient.data.name;

updateClientDetails(newClient.data);

hideDropdown();

} else {

console.error('Failed to create client:', newClient.error);

alert('Failed to create client. Please try again.');

}

} catch (error) {

console.error('Error creating client:', error);

alert('Error creating client. Please try again.');

}

});

}

if (elements.removeClientBtn) {

elements.removeClientBtn.addEventListener('click', function(event) {

event.preventDefault();

if (confirm('Are you sure you want to remove this client from the quote?')) {

['clientIdInput', 'clientNameInput', 'clientEmailInput', 'clientPhoneInput', 'clientAddressInput'].forEach(input => {

if (elements[input]) elements[input].value = '';

});

['clientNamePreview', 'clientEmailPreview', 'clientPhonePreview', 'clientAddressPreview'].forEach(preview => {

if (elements[preview]) elements[preview].textContent = '';

});

elements.clientSearchInput.value = '';

if (elements.clientPreview) elements.clientPreview.classList.add('d-none');

}

});

}

if (elements.editClientBtn) {

elements.editClientBtn.addEventListener('click', async function(event) {

event.preventDefault();

const clientId = elements.clientIdInput.value;

if (clientId) {

try {

const response = await fetch(`/clients/${clientId}`, { cache: 'no-store' });

if (!response.ok) throw new Error('Network response was not ok');

const client = await response.json();

document.getElementById('editClientId').value = client.\_id;

document.getElementById('editClientName').value = client.name;

document.getElementById('editClientEmail').value = client.email;

document.getElementById('editClientPhoneNumber').value = client.phoneNumber;

document.getElementById('editClientStreetAddress').value = client.streetAddress;

document.getElementById('editClientCity').value = client.city;

document.getElementById('editClientState').value = client.state;

document.getElementById('editClientZip').value = client.zip;

const editClientModal = new bootstrap.Modal(document.getElementById('editClientModal'));

editClientModal.show();

} catch (error) {

console.error('Error fetching client for edit:', error);

alert('Error fetching client data. Please try again.');

}

}

});

}

if (elements.editClientForm) {

elements.editClientForm.addEventListener('submit', async function(event) {

event.preventDefault();

const formData = new FormData(elements.editClientForm);

const data = {};

formData.forEach((value, key) => data[key] = value);

try {

const response = await fetch(`/clients/${data.clientId}`, {

method: 'PUT',

headers: {

'Content-Type': 'application/json',

'CSRF-Token': data.\_csrf

},

body: JSON.stringify(data)

});

if (!response.ok) throw new Error('Network response was not ok');

const updatedClient = await response.json();

if (updatedClient.success) {

elements.clientSearchInput.value = updatedClient.data.name;

updateClientDetails(updatedClient.data);

if (elements.clientPreview) elements.clientPreview.classList.remove('d-none');

const editClientModal = bootstrap.Modal.getInstance(document.getElementById('editClientModal'));

editClientModal.hide();

} else {

console.error('Failed to update client:', updatedClient.error);

alert('Failed to update client. Please try again.');

}

} catch (error) {

console.error('Error updating client:', error);

alert('Error updating client. Please try again.');

}

});

}

if (elements.serviceTypeSelect) {

elements.serviceTypeSelect.addEventListener('change', toggleServiceOptions);

toggleServiceOptions(); // Call on page load to set the correct initial state

}

});

```

\*\*Key changes:\*\*

1. Grouped DOM elements into an `elements` object for better organization.

2. Extracted repetitive client detail setting into the `updateClientDetails` function.

3. Simplified null-checks for DOM elements within `updateClientDetails`.

4. Applied `const` and `let` instead of `var`.

5. Included comments to explain logical code blocks.

These changes ensure maintainability and readability while preserving the functionality of your code.

**Main.js**

Your JavaScript code generally looks good and performs the intended task of submitting form data via the Fetch API, handling the response, and redirecting based on the current URL path. However, there are a few improvements and suggestions to make the code more robust, secure, and maintainable.

### Improvements and Suggestions:

1. \*\*Form Validation\*\*:

- Before sending the form data, ensure that all required fields are validated. It will help to catch any client-side errors before making the network request.

2. \*\*Enhanced Error Handling\*\*:

- Provide more informative error messages based on `response.status`. For example, handle specific HTTP status codes differently (e.g., 400 Bad Request, 401 Unauthorized, 500 Internal Server Error).

3. \*\*Security Enhancements\*\*:

- Ensure you sanitize the input data to safeguard against XSS attacks and other vulnerabilities.

- Check if you correctly set up a CSRF token mechanism and validate it on the server side.

4. \*\*Logging\*\*:

- While logging in the console can be useful for development, make sure to remove or conditionally execute logging in production to avoid leaking any sensitive information.

5. \*\*Modular Function\*\*:

- Extract repeated or complex logic into separate functions for better readability and maintainability.

### Improved Code:

Here's the improved version of your script with the mentioned enhancements:

```javascript

document.addEventListener('DOMContentLoaded', function() {

const form = document.querySelector('form');

if (form) {

form.addEventListener('submit', function(event) {

event.preventDefault();

if (!validateForm(form)) {

alert('Please fill out all required fields.');

return;

}

const formData = new FormData(form);

const data = {};

formData.forEach((value, key) => {

data[key] = value;

});

console.log('Form Data:', data); // Remove or conditionally execute this in production

fetch(form.action, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'CSRF-Token': data.\_csrf // Include CSRF token in headers

},

body: JSON.stringify(data)

})

.then(response => {

if (!response.ok) {

return response.json().then(err => Promise.reject({status: response.status, error: err}));

}

return response.json();

})

.then(result => {

console.log('Success:', result);

redirectToProperPage();

})

.catch(({status, error}) => {

console.error(`Failed to submit form (${status}):`, error);

alert(`Failed to submit form. ${error.message || 'Please try again.'}`);

});

});

}

function validateForm(form) {

let isValid = true;

form.querySelectorAll('[required]').forEach(input => {

if (!input.value.trim()) {

isValid = false;

}

});

return isValid;

}

function redirectToProperPage() {

let redirectUrl = '/items';

if (window.location.pathname.includes('/clients')) {

redirectUrl = '/clients';

} else if (window.location.pathname.includes('/quotes')) {

redirectUrl = '/quotes';

}

window.location.href = redirectUrl; // Redirect based on context

}

});

```

### Explanation of Changes:

1. \*\*Form Validation (`validateForm` function)\*\*:

- Added a simple validation function to ensure all required fields are filled out.

2. \*\*Enhanced Error Handling\*\*:

- Refined the `.catch` block to provide more information about the status code and error message.

3. \*\*Sanitized Logging\*\*:

- Added a note to remove or conditionally execute logging in production to avoid leaking sensitive information.

4. \*\*Modularization\*\*:

- Moved the redirection logic to a separate function `redirectToProperPage` for better readability and reusability.

By making these improvements, you will have a more robust, secure, and maintainable codebase.

**QuoteFormSumbussion.js**

The JavaScript code handles dynamic form management for creating quotes with line items within an Express.js/EJS web application. Here are some suggestions for errors, improvements, and best practices:

### Errors & Potential Issues

1. \*\*Handling Multiple Event Bindings:\*\*

- When adding event listeners within `addLineItemRow`, you’re creating a new function for each line item. This can lead to performance issues with a large number of line items. Consider using event delegation where possible.

2. \*\*Error Handling:\*\*

- More error handling around network requests (like `axios.post`) would be beneficial. The current approach catches the error, but you might want to handle different types of errors separately.

3. \*\*Element existence:\*\*

- Ensure that queried elements (such as `subtotalElement`, `taxRateElement`, and `totalElement`) exist to avoid any runtime errors.

### Improvements & Suggestions

1. \*\*Use Template Literals for Readability:\*\*

- Increase readability by using multi-line template literals for the HTML structure inside `lineItem.innerHTML`.

2. \*\*Event Delegation:\*\*

- Consider using event delegation to handle events like clicks for removing line items. This approach is more efficient as you’re adding fewer event listeners.

3. \*\*Separate Concerns:\*\*

- Break down the large `addLineItemRow` function into smaller, more focused functions to adhere to the single-responsibility principle.

4. \*\*Default Error Handling Function:\*\*

- Create a default error handling function to show user-friendly error messages when requests fail.

5. \*\*Enhance User Feedback:\*\*

- Adding loading indicators or success messages could enhance user experience during asynchronous operations.

### Refactored Code

Here's a refactored version implementing some of these suggestions:

```javascript

document.addEventListener('DOMContentLoaded', function() {

const addLineItemBtn = document.getElementById('addLineItemBtn');

const lineItemsContainer = document.getElementById('lineItemsContainer');

const validationErrorsElement = document.getElementById('validationErrors');

const subtotalElement = document.getElementById('subtotal');

const taxRateElement = document.getElementById('taxRate');

const totalElement = document.getElementById('total');

let lineItemCounter = 0;

function createLineItemHTML(id) {

return `

<div class="form-group">

<label for="itemSearch${id}">Item</label>

<input type="text" id="itemSearch${id}" class="form-control itemSearch" placeholder="Search for item...">

<div class="itemDropdown d-none"></div>

</div>

<div class="form-group">

<label for="description${id}">Description</label>

<input type="text" id="description${id}" class="form-control description" placeholder="Description">

</div>

<div class="form-group">

<label for="quantity${id}">Quantity</label>

<input type="number" id="quantity${id}" class="form-control quantity" value="1" min="1">

</div>

<div class="form-group">

<label for="rate${id}">Rate</label>

<input type="number" id="rate${id}" class="form-control rate" value="0" step="0.01">

</div>

<div class="form-group">

<label for="total${id}">Total</label>

<input type="number" id="total${id}" class="form-control total" value="0" readonly>

</div>

<button type="button" class="btn btn-danger removeLineItemBtn">Remove</button>

`;

}

function addLineItemRow() {

lineItemCounter++;

const lineItem = document.createElement('div');

lineItem.classList.add('line-item');

lineItem.innerHTML = createLineItemHTML(lineItemCounter);

lineItemsContainer.appendChild(lineItem);

lineItem.querySelector('.removeLineItemBtn').addEventListener('click', function() {

lineItem.remove();

updatePricing();

});

const quantityInput = lineItem.querySelector('.quantity');

const rateInput = lineItem.querySelector('.rate');

const itemSearchInput = lineItem.querySelector('.itemSearch');

quantityInput.addEventListener('input', updatePricing);

rateInput.addEventListener('input', updatePricing);

itemSearchInput.addEventListener('input', fetchServiceItems);

itemSearchInput.addEventListener('blur', () => setTimeout(() => hideDropdown(lineItem.querySelector('.itemDropdown')), 200));

updatePricing();

}

async function fetchServiceItems(event) {

try {

const query = event.target.value.trim();

const dropdown = event.target.nextElementSibling; // Get the dropdown element

if (query.length > 2) {

const response = await axios.get(`/api/service-items`, { params: { query } });

const serviceItems = response.data.data;

showDropdown(dropdown, event.target, serviceItems);

} else {

hideDropdown(dropdown);

}

} catch (error) {

console.error('Failed to fetch service items:', error);

}

}

function showDropdown(dropdown, inputElement, items) {

dropdown.classList.remove('d-none');

dropdown.innerHTML = '';

items.forEach(item => {

const option = document.createElement('div');

option.className = 'dropdown-item';

option.textContent = item.name;

option.dataset.id = item.\_id;

option.dataset.price = item.price;

option.addEventListener('click', () => selectDropdownItem(option, inputElement));

dropdown.appendChild(option);

});

}

function selectDropdownItem(option, inputElement) {

const lineItem = inputElement.closest('.line-item');

inputElement.value = option.textContent;

lineItem.querySelector('.rate').value = option.dataset.price;

lineItem.querySelector('.description').value = option.description || '';

let itemIdInput = lineItem.querySelector('input[name="serviceItemIds[]"]');

if (!itemIdInput) {

itemIdInput = document.createElement('input');

itemIdInput.type = 'hidden';

itemIdInput.name = 'serviceItemIds[]';

lineItem.appendChild(itemIdInput);

}

itemIdInput.value = option.dataset.id;

updatePricing();

hideDropdown(inputElement.nextElementSibling);

}

function hideDropdown(dropdown) {

dropdown.classList.add('d-none');

dropdown.innerHTML = '';

}

function updatePricing() {

let subtotal = 0;

document.querySelectorAll('.line-item').forEach(lineItem => {

const quantity = parseFloat(lineItem.querySelector('.quantity').value) || 0;

const rate = parseFloat(lineItem.querySelector('.rate').value) || 0;

const total = quantity \* rate;

lineItem.querySelector('.total').value = total.toFixed(2);

subtotal += total;

});

subtotalElement.textContent = `$${subtotal.toFixed(2)}`;

const taxRate = parseFloat(taxRateElement.value) || 0;

const taxAmount = subtotal \* (taxRate / 100);

const total = subtotal + taxAmount;

totalElement.textContent = `$${total.toFixed(2)}`;

}

addLineItemBtn.addEventListener('click', addLineItemRow);

taxRateElement.addEventListener('input', updatePricing);

document.getElementById('quoteForm').addEventListener('submit', async (e) => {

e.preventDefault();

const formData = new FormData(e.target);

const data = Object.fromEntries(formData.entries());

data.serviceItems = [];

let formValid = validateForm(data);

if (!formValid) {

validationErrorsElement.innerHTML = 'Please fill out all required fields for service items.';

validationErrorsElement.classList.remove('d-none');

return;

}

data.clientName = validateClientName();

if (!data.clientName) {

validationErrorsElement.innerHTML = 'Please select a client.';

validationErrorsElement.classList.remove('d-none');

return;

}

data.frequency = validateFrequency(data.serviceType);

if (data.frequency === false) {

validationErrorsElement.innerHTML = 'Please select a valid frequency.';

validationErrorsElement.classList.remove('d-none');

return;

}

console.log('Form data being sent:', data);

try {

const response = await axios.post('/quotes', data, {

headers: {

'Content-Type': 'application/json',

'X-CSRF-Token': formData.get('\_csrf')

}

});

console.log('Quote created successfully', response.data);

window.location.href = '/dashboard';

} catch (error) {

console.error('Error creating quote', error.response?.data, error.message, error.stack);

validationErrorsElement.innerHTML = 'Failed to create quote. Please try again.';

validationErrorsElement.classList.remove('d-none');

}

});

function validateForm(data) {

let formValid = true;

document.querySelectorAll('.line-item').forEach(lineItem => {

const itemSearch = lineItem.querySelector('.itemSearch').value.trim();

const description = lineItem.querySelector('.description').value.trim();

const quantity = parseFloat(lineItem.querySelector('.quantity').value) || 0;

const rate = parseFloat(lineItem.querySelector('.rate').value) || 0;

const serviceItemId = lineItem.querySelector('input[name="serviceItemIds[]"]')?.value;

if (itemSearch && quantity && rate && serviceItemId) {

data.serviceItems.push({

serviceItemId,

description,

quantity,

rate

});

} else {

formValid = false;

}

});

return formValid;

}

function validateClientName() {

const clientName = document.getElementById('clientSearch').value;

return clientName || null;

}

function validateFrequency(serviceType) {

if (serviceType === 'Recurring') {

const frequency = document.getElementById('frequency').value;

return frequency !== 'Choose...' ? frequency : false;

}

return null;

}

// Initial line item row

addLineItemRow();

});

```

### Summary of Changes

- \*\*Template Literals:\*\* Used for clearer HTML structure within JavaScript.

- \*\*Separated\*\* smaller functions for better readability and maintenance.

- \*\*Event Delegation:\*\* Suggested approach for event handling.

- \*\*Asynchronous Handling:\*\* Added structured try-catch blocks around asynchronous code.

- \*\*Validation Functions:\*\* Improved parallel validation for different form parts.

**~~ServiceItemsManagement.js~~**

Here's a thorough review of your code with suggestions for improvement and error fixes:

### Key Observations:

1. \*\*Basic Functionality\*\*: The code aims to manage service items by fetching them from a server and enabling addition and deletion operations.

2. \*\*Event Handling and Async Operations\*\*: Utilizes `fetch` API for async operations along with proper error handling.

3. \*\*Dynamic DOM Manipulation\*\*: Sensibly adds new rows to a table and sets up event listeners to handle item deletions.

### Issues & Improvements:

1. \*\*Error Logging Within Success Path\*\*:

- The `logToServer('error', `Error fetching service items: ${error.message}`);` statement should be moved within the catch block of `fetchAndPopulateServiceItems()`. It currently resides in the success path and will always log an error message even if the fetch was successful.

2. \*\*Missing `addServiceItemBtn` and Its Event Listener\*\*:

- Currently, there is a reference to `addServiceItemBtn` but it isn't used for any event. This could be improved by adding functionality for dynamically adding new service items.

3. \*\*CSRF Token Handling\*\*:

- The CSRF token is being directly fetched from the document once per delete operation. This is good practice but ensure that this token field is present in your HTML form.

4. \*\*Data Formats\*\*:

- Consider formatting the price with `toFixed(2)` to ensure it shows two decimal places ($100.00 instead of $100).

5. \*\*Error Handling Visibility\*\*:

- Currently, errors are logged to the console. For a production environment, consider providing user feedback in the UI.

6. \*\*Potential Performance Improvements\*\*:

- Consider batch DOM updates if the initial load contains a large number of items, this reduces the reflows and repaints.

Here's the adjusted version of the code considering the suggestions and improvements:

```javascript

document.addEventListener('DOMContentLoaded', function() {

const serviceItemsList = document.getElementById('serviceItemsList');

const addServiceItemBtn = document.getElementById('addServiceItemBtn');

// Function to add a new item to the list

function addServiceItem(itemId, name, category, price, description) {

const serviceItemElement = document.createElement('tr');

serviceItemElement.classList.add('item');

serviceItemElement.dataset.itemId = itemId; // Assign itemId to the dataset for edit and delete operations

serviceItemElement.innerHTML = `

<td>${name}</td>

<td>${category}</td>

<td>${description}</td>

<td>$${price.toFixed(2)}</td>

<td>

<a href="/items/${itemId}/edit" class="btn btn-warning btn-sm">Edit</a>

<button class="btn btn-danger btn-sm deleteServiceItemBtn">Delete</button>

</td>

`;

serviceItemsList.appendChild(serviceItemElement);

// Event listener for delete button

serviceItemElement.querySelector('.deleteServiceItemBtn').addEventListener('click', async function() {

try {

const response = await fetch(`/items/${itemId}`, {

method: 'DELETE',

headers: {

'Content-Type': 'application/json',

'X-CSRF-Token': document.querySelector('input[name="\_csrf"]').value // Ensure CSRF token is included

}

});

if (response.ok) {

serviceItemElement.remove();

console.log(`Service item ${itemId} deleted.`);

} else {

console.error('Failed to delete service item:', response.statusText);

}

} catch (error) {

console.error('Failed to delete service item:', error);

}

});

}

// Fetch and populate service items from the server

async function fetchAndPopulateServiceItems() {

try {

const response = await fetch('/items'); // Correct endpoint

const serviceItems = await response.json();

serviceItems.forEach(item => {

addServiceItem(item.\_id, item.name, item.category, item.price, item.description);

});

console.log('Service items fetched and populated successfully.');

} catch (error) {

console.error('Failed to fetch service items:', error);

logToServer('error', `Error fetching service items: ${error.message}`);

}

}

// Initial fetch and populate service items

fetchAndPopulateServiceItems();

// Example event listener for adding a new service item

if (addServiceItemBtn) {

addServiceItemBtn.addEventListener('click', function() {

// Logic for adding a new service item (possibly opening a form/modal for input)

});

}

// Additional logic for handling form submissions, etc., can be added here

});

```

### Summary:

The above adjustments aim to correct observed errors, enhance the code's robustness, and add missing functionality to increase overall maintainability and scalability. Implementing these changes will make your code cleaner, and user experience more fluid while adding new items.

**~~Settings.js~~**

Analyzing the provided JavaScript code for an Express.js or EJS environment, the code seems quite comprehensive and functional for handling form submissions and user settings. However, there are a few areas where improvements could be made to enhance the functionality, error handling, and code readability.

### Error Analysis

No evident functional errors are found in the logic. The script appears to:

1. Fetch user information on page load.

2. Populate the form fields with the fetched data.

3. Submit the settings form.

4. Submit the change password form.

5. Update the CSRF Token if a new one is received from the server.

### Improvements and Suggestions

#### 1. Reuse and Modularize Code

There are similarities between the form submission handlers that could be modularized to avoid code duplication.

```javascript

// Function to handle form submission

async function handleFormSubmission(url, formData) {

try {

const response = await fetch(url, {

method: 'POST',

body: formData,

credentials: 'include',

headers: {

'CSRF-Token': formData.get('\_csrf') // Ensure CSRF token is included in the request headers

}

});

if (!response.ok) {

throw new Error('Request failed');

}

const data = await response.json();

if (data.success) {

alert('Operation completed successfully.');

// Update CSRF token if a new one is provided

if (data.csrfToken) {

updateCsrfToken(data.csrfToken);

}

} else {

alert(data.message || 'Operation failed.');

}

} catch (error) {

console.error('Error:', error.message);

alert('Error occurred during the operation.');

}

}

```

Then call this function in the event listeners:

```javascript

// Handle user information form submission

if (settingsForm) {

settingsForm.addEventListener('submit', function(e) {

e.preventDefault();

const formData = new FormData(this);

handleFormSubmission('/settings', formData);

});

}

// Handle password change form submission

if (changePasswordForm) {

changePasswordForm.addEventListener('submit', function(e) {

const newPassword = document.querySelector('#newPassword').value;

const confirmNewPassword = document.querySelector('#confirmNewPassword').value;

if (newPassword !== confirmNewPassword) {

e.preventDefault();

alert('New passwords do not match.');

return;

}

const formData = new FormData(this);

handleFormSubmission('/settings/change-password', formData);

});

}

```

#### 2. Improved Error Handling and Logging

Adding more descriptive and user-friendly error messages can improve debugging and user experience.

```javascript

catch (error) {

console.error('Error:', error.message);

alert('An unexpected error occurred. Please try again later.');

}

```

#### 3. Enhance Security

Sanitize the form data before submitting it to avoid XSS attacks.

```javascript

function sanitizeInput(input) {

const div = document.createElement('div');

div.appendChild(document.createTextNode(input));

return div.innerHTML;

}

```

Use this function to sanitize each field before submission:

```javascript

function sanitizeFormData(formData) {

for (let [key, value] of formData.entries()) {

formData.set(key, sanitizeInput(value));

}

return formData;

}

settingsForm.addEventListener('submit', function(e) {

e.preventDefault();

let formData = new FormData(this);

formData = sanitizeFormData(formData);

handleFormSubmission('/settings', formData);

});

```

#### 4. Accessibility Improvements

Ensure labels are connected to inputs using `for` attribute to improve accessibility.

```html

<label for="businessName">Business Name</label>

<input id="businessName" name="businessName">

```

#### 5. Performance Optimization

Consider caching the fetched user data, or debouncing input events if needed, to improve performance.

```javascript

let userCache = null;

async function fetchUserInfo() {

if (userCache) {

return userCache;

}

...

userCache = data;

}

```

#### 6. Validation and Feedback

Add form validation feedback to aid user correction and experience.

```javascript

function validateSettingsForm(formData) {

let isValid = true;

// Example validation for email

const email = formData.get('email');

if (!email.includes('@')) {

isValid = false;

document.querySelector('#email').classList.add('invalid');

alert('Please provide a valid email address.');

}

return isValid;

}

settingsForm.addEventListener('submit', function(e) {

e.preventDefault();

const formData = new FormData(this);

if (validateSettingsForm(formData)) {

handleFormSubmission('/settings', formData);

}

});

```

### Conclusion

The provided code functions correctly but can benefit from the above-mentioned improvements to enhance security, performance, and maintainability. Modularizing repetitive code, improving error handling, sanitizing inputs, and implementing form validation are key areas to focus on.

**apiRoutes.js**

The provided JavaScript code snippet for defining routes in an Express.js application looks generally correct and follows good practices. However, there are always areas for improvement and ensuring robustness. Here's an analysis along with some suggestions for potential improvements:

### Analysis

1. \*\*Route Definition:\*\*

- `GET /service-items`: This route correctly uses the `isAuthenticated` middleware to enforce authentication before calling the `getAllServiceItems` controller function.

2. \*\*Modularization:\*\*

- The code uses modularization effectively by separating concerns between routing, middleware, and controllers.

3. \*\*Exporting Router:\*\*

- The router object is exported correctly, making it available for use in the main application file.

### Potential Improvements

1. \*\*Error Handling in Middleware and Controller:\*\*

- Ensure that both `isAuthenticated` and `getAllServiceItems` have proper error handling mechanisms. For example, they should call `next(err)` where `err` is an error object when something goes wrong, to allow the error-handling middleware to handle it.

2. \*\*Adding Comments:\*\*

- While the comment `// Get all service items` is helpful, you could provide more comprehensive comments if needed (especially for complex logic).

3. \*\*Security Considerations:\*\*

- Make sure your `isAuthenticated` middleware provides thorough checks and balances to prevent unauthorized access efficiently.

4. \*\*Request Validation:\*\*

- If applicable, consider adding request validation to check any parameters or query strings.

5. \*\*Asynchronous Handling:\*\*

- Ensure `getAllServiceItems` handles asynchronous operations (e.g., DB calls) correctly, probably using `async`/`await` and try/catch blocks.

### Sample Error Handling in Middleware and Controller

Here's a quick example of what robust error handling could look like inside the middleware and controller:

#### Middleware (authMiddleware.js)

```javascript

const isAuthenticated = (req, res, next) => {

try {

// Assuming that user authentication will set req.user

if (req.user) {

return next();

} else {

return res.status(401).json({ error: 'Unauthorized' });

}

} catch (err) {

next(err); // Pass the error to the error-handling middleware

}

};

module.exports = { isAuthenticated };

```

#### Controller (serviceItemController.js)

```javascript

const { getServiceItemsFromDB } = require('../models/serviceItemModel'); // Hypothetical data access layer

const getAllServiceItems = async (req, res, next) => {

try {

const serviceItems = await getServiceItemsFromDB(); // Hypothetical async DB function

res.json(serviceItems);

} catch (err) {

next(err); // Pass the error to the error-handling middleware

}

};

module.exports = { getAllServiceItems };

```

### Conclusion

The provided code is on the right track and should work correctly in a standard Express.js application. By ensuring comprehensive error handling and validation in your middleware and controller functions, you can improve the robustness and maintainability of your code.

Would you like further details on any specific part or additional improvements related to performance, logging, or request validation?

**~~authRoutes.js~~**

The provided JavaScript code defines Express.js routes for user authentication, including registration, login, and logout functionalities. Several suggestions and improvements can be made to enhance the robustness, readability, and security of this code.

### Improvements and Suggestions

1. \*\*Password Hashing:\*\*

- Ensure that the password is hashed before saving it to the database.

- Typically, this is performed using libraries such as `bcryptjs`.

2. \*\*Validation and Sanitization:\*\*

- Validate and sanitize user inputs to protect against injection attacks and invalid data.

- Libraries like `express-validator` can be used for this purpose.

3. \*\*Error Handling:\*\*

- Provide more informative error messages for the end-users but be cautious not to expose sensitive details.

- Differentiate between various error statuses, e.g., `400` for bad request, `401` for unauthorized, etc.

4. \*\*Async Error Handling:\*\*

- Ensure async middleware errors are properly caught and passed to the next error-handling middleware.

5. \*\*Security Headers and CSRF:\*\*

- While CSRF protection is in place, consider adding security headers using a library like `helmet`.

6. \*\*Redirection on Successful Logout:\*\*

- Ensure proper handling if the user is already logged out and attempts to log out again.

7. \*\*Session Handling:\*\*

- Ensure using secure session management practices, such as using HTTPS and setting secure session cookie flags.

Here's the revised code incorporating these improvements:

```javascript

const express = require('express');

const User = require('../models/User');

const csrf = require('csurf');

const bcrypt = require('bcryptjs');

const { isAuthenticated } = require('../middleware/authMiddleware');

const { body, validationResult } = require('express-validator');

const router = express.Router();

// Setup CSRF protection middleware

const csrfProtection = csrf({ cookie: true });

router.get('/register', csrfProtection, (req, res) => {

res.render('register', { csrfToken: req.csrfToken() });

});

router.post('/register',

csrfProtection,

// Validate and sanitize input fields

[

body('username').isLength({ min: 3 }).trim().escape(),

body('email').isEmail().normalizeEmail(),

body('password').isLength({ min: 6 }).trim().escape(),

body('businessName').optional().trim().escape(),

body('name').optional().trim().escape(),

body('phoneNumber').optional().trim().escape(),

body('streetAddress').optional().trim().escape(),

body('city').optional().trim().escape(),

body('state').optional().trim().escape(),

body('zipCode').optional().trim().escape()

],

async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).render('register', {

csrfToken: req.csrfToken(),

errors: errors.array()

});

}

try {

const { username, password, businessName, name, phoneNumber, email, streetAddress, city, state, zipCode } = req.body;

// Hash the password before saving

const hashedPassword = await bcrypt.hash(password, 12);

const newUser = new User({

username,

password: hashedPassword,

businessName,

name,

phoneNumber,

email,

streetAddress,

city,

state,

zipCode

});

await newUser.save();

res.redirect('/auth/login');

} catch (error) {

console.error('Registration error:', error);

res.status(500).send('Error registering new user.');

}

});

router.get('/login', csrfProtection, (req, res) => {

res.render('login', { csrfToken: req.csrfToken() });

});

router.post('/login',

csrfProtection,

[

body('username').trim().escape(),

body('password').trim().escape()

],

async (req, res) => {

try {

const { username, password } = req.body;

const user = await User.findOne({ username });

if (!user) {

console.warn(`User not found: ${username}`);

return res.status(401).send('Invalid username or password');

}

const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) {

console.warn(`Invalid password for user: ${username}`);

return res.status(401).send('Invalid username or password');

}

req.session.userId = user.\_id;

res.redirect('/');

} catch (error) {

console.error('Login error:', error);

res.status(500).send('An error occurred during login');

}

});

router.get('/logout', isAuthenticated, (req, res) => {

if (!req.session) {

return res.redirect('/auth/login');

}

req.session.destroy(err => {

if (err) {

console.error('Error during session destruction:', err);

return res.status(500).send('Error logging out');

}

res.clearCookie('connect.sid');

res.redirect('/auth/login');

});

});

module.exports = router;

```

\*\*Summary of Changes:\*\*

- Added password hashing using `bcryptjs`.

- Added validation and sanitization for user inputs using `express-validator`.

- Enhanced error handling and custom error messages.

- Handled session destruction more gracefully.

- Secured the session destruction process by clearing cookies after logging out.

**clientRoutes.js**

Your JavaScript code for handling client routes using Express.js appears to be well-structured and follows common best practices for middleware usage, authentication, and CSRF protection. Here are a few suggestions and improvements for optimization and error handling:

### Code Suggestions and Improvements

1. \*\*Redundant Code Removal:\*\*

- The code included twice is unnecessary. The file contents only need to be displayed once.

2. \*\*Consistent Log Messages:\*\*

- Maintain consistent and clear log messages for better debugging.

3. \*\*Improved Error Handling:\*\*

- Consider adding more specific error messages and handling various scenarios.

4. \*\*Encapsulation of Common Middlewares:\*\*

- When repeating middleware sequences, consider encapsulating them to reduce redundancy.

5. \*\*Improved Response Codes and Messages:\*\*

- Be more specific with error response codes and messages to aid in frontend debugging.

6. \*\*Enhanced Security:\*\*

- Consider explicitly specifying allowed HTTP methods for better security.

Here's the updated version of your `clientRoutes.js` file with these improvements:

```javascript

const express = require('express');

const router = express.Router();

const Client = require('../models/clientModel');

const { createClient, searchClients, updateClient, deleteClient } = require('../controllers/clientController');

const { isAuthenticated } = require('../middleware/authMiddleware');

const csrf = require('csurf');

const getClient = require('../middleware/getClient');

// CSRF Protection Middleware

const csrfProtection = csrf({ cookie: true });

// Middleware Sequence for Routes

const authAndCsrf = [isAuthenticated, csrfProtection];

// Get all clients and render the clients.ejs template

router.get('/', ...authAndCsrf, async (req, res) => {

try {

const clients = await Client.find();

res.render('clients', { clients, csrfToken: req.csrfToken() });

console.log("Fetched all clients successfully.");

} catch (err) {

console.error(`Error fetching clients: ${err.message}`, err.stack);

res.status(500).json({ message: 'Failed to fetch clients. Please try again later.' });

}

});

// Render the form to create a new client

router.get('/new', ...authAndCsrf, (req, res) => {

console.log("Rendering createClient.ejs");

res.render('createClient', { csrfToken: req.csrfToken() });

});

// Search for clients

router.get('/search', ...authAndCsrf, searchClients);

// Get one client

router.get('/:id', ...authAndCsrf, getClient, (req, res) => {

res.status(200).json(req.client);

});

// Render the form to edit a client

router.get('/:id/edit', ...authAndCsrf, getClient, (req, res) => {

res.render('editClient', { client: req.client, csrfToken: req.csrfToken() });

console.log(`Rendered editClient.ejs for client ID: ${req.params.id}`);

});

// Create new client

router.post('/', ...authAndCsrf, createClient);

// Update client

router.put('/:id', ...authAndCsrf, getClient, updateClient);

// Delete client

router.delete('/:id', ...authAndCsrf, getClient, deleteClient);

module.exports = router;

```

### Explanation of Improvements

1. \*\*Encapsulation of Middleware:\*\*

- Using the spread operator (`...`) to apply multiple middlewares in a concise manner reduces redundancy.

2. \*\*Improved Error Messages:\*\*

- Provided general error messages to inform the user about the failure while logging the actual error for debugging.

3. \*\*Consistent Logging:\*\*

- Ensured all log statements follow a similar format for consistency.

4. \*\*Redundant Code Removed:\*\*

- Removed the duplicate block of code to keep the file clean and maintainable.

By incorporating these improvements, you enhance the code quality, maintainability, and user experience, while maintaining security best practices.

**logRoutes.js**

The provided JavaScript file `logRoutes.js` for handling log routes using Express.js is generally well-structured. However, there are a few improvements, suggestions, and error-checking points that can be addressed to make the code more robust and maintainable.

### 1. Ensure JSON Parsing Middleware

Make sure that the main application file (e.g., `app.js` or `server.js`) includes middleware to parse JSON request bodies. Otherwise, `req.body` might be `undefined`.

```javascript

app.use(express.json());

```

### 2. Validate Log Levels

Validate the `level` field to ensure it matches expected log levels (e.g., 'info', 'error', 'warn').

### 3. Use HTTP Status Codes Constants

Although express doesn't provide constants for status codes, using constants defined in a separate file can help.

### 4. Add Input Sanitization

Sanitize inputs to prevent log injection or any form of malicious data.

### 5. Handle Logger Errors

Handle errors that may occur from the logging operation to avoid unhandled promise rejections or application crashes.

### Improved `logRoutes.js`

```javascript

const express = require('express');

const router = express.Router();

const logger = require('../logger'); // Ensure logger is correctly implemented and handles errors

const { check, validationResult } = require('express-validator'); // Import express-validator for validation

// Predefined log levels, depending on your logging library (e.g., winston)

const VALID\_LOG\_LEVELS = ['info', 'warn', 'error', 'debug'];

router.post(

'/',

[

check('level')

.isIn(VALID\_LOG\_LEVELS)

.withMessage(`Level must be one of: ${VALID\_LOG\_LEVELS.join(', ')}`),

check('message')

.not()

.isEmpty()

.withMessage('Message should not be empty')

],

(req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

const { level, message } = req.body;

// Handle logger errors (let's assume logger.log is async)

try {

logger.log({ level, message });

res.status(200).send('Log received');

} catch (err) {

res.status(500).send('Error logging message');

}

}

);

module.exports = router;

```

### Additional Considerations:

1. \*\*CORS\*\*: Ensure that cross-origin resource sharing is properly configured if the client application is hosted on a different domain.

2. \*\*Security Headers\*\*: Make sure you're using security-related HTTP headers (e.g., using helmet middleware).

3. \*\*Rate Limiting\*\*: Implement rate limiting to avoid abuse of the logging endpoint.

4. \*\*Error Logging\*\*: Log errors properly for debugging purposes.

Improved input validation and error handling will make the application more secure and robust to potential misuse.

**quoteRoutes.js**

Your code is generally well-structured, but there are a few areas for improvement and considerations. Below are some potential errors, improvements, and suggestions:

### Potential Errors:

1. \*\*Duplicate PUT Route Logic\*\*:

- Both the `POST` and `PUT` routes for creating and updating quotes respectively contain the same logic. This could be an unintentional duplication, especially for the `PUT` route which updates an existing quote.

2. \*\*Missing CSRF Token in JSON Response for `quote.GET`\*\*:

- When `req.accepts('html')` is false, your response for `GET /:id` does not return a CSRF token in the JSON output. This could cause issues for clients expecting this token for further interactions.

### Improvements and Suggestions:

1. \*\*Code DRYness\*\*:

- Avoid code duplication between `POST /` and `PUT /:id`. Abstract the quote creation logic into a separate function to be reused.

2. \*\*Consistent Error Handling\*\*:

- Use a standardized way to handle and format errors. This can make debugging and client-side error handling easier.

3. \*\*File Storage Directory Management\*\*:

- Ensure the `uploads/` directory exists or handle cases where the directory might not exist, for instance, by using the `fs` module to check and create the directory if it doesn't exist.

4. \*\*Centralized Configuration for CSRF Protection\*\*:

- If you use CSRF protection extensively, consider configuring it centrally, possibly in the main application file or a middleware.

5. \*\*Security Considerations\*\*:

- Validate and sanitize user inputs more thoroughly to enhance security, especially for fields like `title`, `scopeOfWork`, etc.

6. \*\*Logging\*\*:

- Using a logging library like `winston` instead of `console.log` for better control over logging levels and formats.

### Revised Code Skeleton:

Below, I've made some refactoring and added small improvements for the discussed points:

```javascript

const express = require('express');

const router = express.Router();

const Quote = require('../models/quoteModel');

const Client = require('../models/clientModel');

const ServiceItem = require('../models/serviceItem');

const User = require('../models/User');

const { sendQuoteDataToCRM } = require('../utils/crmIntegration');

const csrf = require('csurf');

const { isAuthenticated } = require('../middleware/authMiddleware');

const quoteController = require('../controllers/quoteController');

const multer = require('multer');

const path = require('path');

const fs = require('fs');

const winston = require('winston');

// Set up logging

const logger = winston.createLogger({

level: 'info',

format: winston.format.json(),

transports: [

new winston.transports.Console(),

new winston.transports.File({ filename: 'combined.log' })

]

});

// Ensure the uploads directory exists

const uploadsDirectory = 'uploads/';

if (!fs.existsSync(uploadsDirectory)) {

fs.mkdirSync(uploadsDirectory);

}

// Configure storage for uploaded files

const storage = multer.diskStorage({

destination: function (req, file, cb) {

cb(null, uploadsDirectory);

},

filename: function (req, file, cb) {

cb(null, Date.now() + path.extname(file.originalname));

}

});

// Initialize multer with storage configuration

const upload = multer({ storage: storage });

// Function to calculate the subtotal

const calculateSubtotal = (serviceItems) => {

return serviceItems.reduce((sum, item) => sum + (item.customPrice || item.rate) \* item.quantity, 0);

};

// CSRF Protection Middleware using csurf

const csrfProtection = csrf({ cookie: true });

const processQuote = async (req, res) => {

const { clientId, clientName, title, scopeOfWork, serviceType, frequency, initialCleaningOptions, serviceItems, userAddress, distance, taxRate } = req.body;

// Validate service items

if (!serviceItems || !Array.isArray(serviceItems) || serviceItems.length === 0) {

return res.status(400).json({ error: 'Service items are required.' });

}

const processedServiceItems = serviceItems.map(item => ({

serviceItemId: item.serviceItemId,

description: item.description,

quantity: item.quantity,

customPrice: item.customPrice ? item.customPrice : item.rate

}));

const subtotal = calculateSubtotal(processedServiceItems);

const calculatedTaxRate = parseFloat(taxRate) || 7.5;

const total = subtotal + (subtotal \* (calculatedTaxRate / 100));

const attachments = (req.files['attachments'] || []).map(file => ({

savedFilename: file.filename,

originalFilename: file.originalname

}));

const contracts = (req.files['contracts'] || []).map(file => ({

savedFilename: file.filename,

originalFilename: file.originalname

}));

return new Quote({

clientId,

clientName,

title,

scopeOfWork,

serviceType,

frequency,

initialCleaningOptions,

serviceItems: processedServiceItems,

userAddress,

distance,

subtotal,

taxRate: calculatedTaxRate,

total,

attachments,

contracts,

});

};

// GET route for retrieving and rendering all quotes

router.get('/', isAuthenticated, csrfProtection, async (req, res) => {

try {

const quotes = await Quote.find().populate('clientId');

logger.info(`Rendering quotes.ejs with quotes data, count: ${quotes.length}`);

res.render('quotes', { quotes, csrfToken: req.csrfToken() });

} catch (error) {

logger.error(`Error retrieving quotes: ${error.message}`, error.stack);

res.status(500).json({ error: error.message });

}

});

// Route to render the form to create a new quote

router.get('/new', isAuthenticated, csrfProtection, async (req, res) => {

try {

const clients = await Client.find();

const serviceItems = await ServiceItem.find();

const userId = req.session.userId;

const user = await User.findById(userId);

res.render('createQuote', { clients, serviceItems, user, csrfToken: req.csrfToken() });

} catch (error) {

logger.error(`Error fetching data for quote creation: ${error.message}`, error.stack);

res.status(500).json({ error: error.message });

}

});

// GET route to render the form to edit an existing quote

router.get('/:id/edit', isAuthenticated, csrfProtection, async (req, res) => {

try {

const quote = await Quote.findById(req.params.id).populate('clientId').populate('serviceItems.serviceItemId');

if (!quote) {

logger.info(`Quote not found with ID: ${req.params.id}`);

return res.status(404).json({ error: 'Quote not found' });

}

const clients = await Client.find();

const serviceItems = await ServiceItem.find();

const userId = req.session.userId;

const user = await User.findById(userId);

res.render('editQuote', { quote, clients, serviceItems, user, csrfToken: req.csrfToken() });

} catch (error) {

logger.error(`Error fetching data for quote editing: ${error.message}`, error.stack);

res.status(500).json({ error: error.message });

}

});

// POST route for creating a new quote with CSRF protection

router.post('/', isAuthenticated, upload.fields([{ name: 'attachments' }, { name: 'contracts' }]), csrfProtection, async (req, res) => {

logger.info('Request body:', req.body);

logger.info('Uploaded files:', req.files);

try {

const newQuote = await processQuote(req, res);

await newQuote.save();

logger.info('Quote saved successfully:', newQuote);

res.status(201).json(newQuote);

} catch (error) {

logger.error(`Error creating new quote: ${error.message}`, error.stack);

res.status(400).json({ error: `Failed to create quote. ${error.message}` });

}

});

// GET route for retrieving a single quote by ID

router.get('/:id', isAuthenticated, csrfProtection, async (req, res) => {

try {

const quote = await Quote.findById(req.params.id)

.populate('clientId')

.populate({ path: 'serviceItems.serviceItemId', model: 'ServiceItem' });

if (!quote) {

logger.info(`Quote not found with ID: ${req.params.id}`);

return res.status(404).json({ error: 'Quote not found' });

}

logger.info(`Retrieved quote with ID: ${req.params.id}`);

if (req.accepts('html')) {

res.render('quoteDetails', { quote, csrfToken: req.csrfToken() });

} else {

res.status(200).json({ quote, csrfToken: req.csrfToken() });

}

} catch (error) {

logger.error(`Error retrieving quote with ID ${req.params.id}: ${error.message}`, error.stack);

res.status(500).json({ error: error.message });

}

});

// PUT route for updating a quote by its ID with CSRF protection

router.put('/:id', isAuthenticated, upload.fields([{ name: 'attachments' }, { name: 'contracts' }]), csrfProtection, async (req, res) => {

logger.info('Request body:', req.body);

logger.info('Uploaded files:', req.files);

try {

const quote = await processQuote(req, res);

quote.\_id = req.params.id; // Set the ID to match the existing quote

await Quote.findByIdAndUpdate(req.params.id, quote);

logger.info('Quote updated successfully:', quote);

res.status(200).json(quote);

} catch (error) {

logger.error(`Error updating quote: ${error.message}`, error.stack);

res.status(400).json({ error: `Failed to update quote. ${error.message}` });

}

});

// DELETE route for removing a quote with CSRF protection

router.delete('/:id', isAuthenticated, csrfProtection, async (req, res) => {

try {

const deletedQuote = await Quote.findByIdAndDelete(req.params.id);

if (!deletedQuote) {

logger.info(`Quote not found for deletion with ID: ${req.params.id}`);

return res.status(404).json({ error: 'Quote not found' });

}

logger.info(`Deleted quote with ID: ${req.params.id}`);

res.status(204).send();

} catch (error) {

logger.error(`Error deleting quote with ID ${req.params.id}: ${error.message}`, error.stack);

res.status(500).json({ error: error.message });

}

});

// POST route to calculate distance

router.post('/calculate-distance', isAuthenticated, csrfProtection, quoteController.calculateDistance);

module.exports = router;

```

### Summary of Key Changes:

- \*\*Logging with `winston`\*\* for better logging control.

- \*\*Directory Check\*\* for uploads directory existence.

- \*\*Abstracted Quote Processing Logic\*\* to `processQuote` function to avoid duplication.

- \*\*CSRF Token Addition\*\* in JSON response for `GET /:id`.

- Consistent structure and better error handling across all routes.

These changes should improve maintainability, readability, and consistency of your code.

**~~serviceItemRoutes.js~~**

Your code, for the most part, is structured well and logically. However, there are a few issues and potential improvements that can be pointed out:

### Issues:

1. \*\*Misplaced `deleteServiceItem` Function\*\*:

- This function should be part of the controller (`serviceItemController.js`) and not directly in the routes file.

2. \*\*Missing `ServiceItem` Model\*\*:

- The code references a `ServiceItem` model, but it is not imported anywhere.

3. \*\*Improper HTTP Method for Deletion in Route\*\*:

- The actual API route for deleting a service item is missing from the router definition.

### Improvements and Suggestions:

1. \*\*Consistency in Error Handling\*\*:

- Consistently render error pages or send JSON responses in a structured format.

2. \*\*Route Structure\*\*:

- Define all required CRUD operations clearly in your routes.

3. \*\*Use Async/Await Properly\*\*:

- Ensure that your asynchronous functions are properly awaited and that the error handling is consistent across all routes.

Here's the corrected and improved version of the code:

### serviceItemRoutes.js

```javascript

const express = require('express');

const router = express.Router();

const { createServiceItem, getServiceItemById, updateServiceItem, deleteServiceItem, renderServiceItemsPage } = require('../controllers/serviceItemController');

const csrf = require('csurf');

const { isAuthenticated } = require('../middleware/authMiddleware');

// Setup CSRF protection middleware

const csrfProtection = csrf({ cookie: true });

// GET route to fetch all service items and render the manage service items page

router.get('/', isAuthenticated, csrfProtection, renderServiceItemsPage);

// GET route to render the create item page

router.get('/new', isAuthenticated, csrfProtection, (req, res) => {

res.render('createItem', { csrfToken: req.csrfToken() });

});

// POST route to create a new service item

router.post('/', isAuthenticated, csrfProtection, async (req, res) => {

try {

await createServiceItem(req, res);

} catch (error) {

console.error(`Error creating service item: ${error.message}`, error);

res.status(400).render('createItem', { error: 'Failed to create item', csrfToken: req.csrfToken() });

}

});

// GET route to render the edit item page

router.get('/:id/edit', isAuthenticated, csrfProtection, async (req, res) => {

try {

const item = await getServiceItemById(req.params.id);

res.render('editItem', { item, csrfToken: req.csrfToken() });

} catch (error) {

console.error(`Error fetching service item for editing: ${error.message}`, error);

res.status(500).send('Error fetching service item for editing');

}

});

// PUT route to update a service item by ID

router.put('/:id', isAuthenticated, csrfProtection, async (req, res) => {

try {

await updateServiceItem(req, res);

} catch (error) {

console.error(`Error updating service item: ${error.message}`, error);

res.status(400).send('Error updating service item');

}

});

// DELETE route to delete a service item by ID

router.delete('/:id', isAuthenticated, csrfProtection, async (req, res) => {

try {

await deleteServiceItem(req, res);

} catch (error) {

console.error(`Error deleting service item: ${error.message}`, error);

res.status(400).render('error', { message: 'Failed to delete item', csrfToken: req.csrfToken() });

}

});

module.exports = router;

```

### serviceItemController.js

Ensure that your controller has the reference to the `ServiceItem` model and properly structured CRUD operations.

```javascript

const ServiceItem = require('../models/ServiceItem'); // Import the ServiceItem model

exports.createServiceItem = async (req, res) => {

// your code for creating a service item

};

exports.getServiceItemById = async (id) => {

// your code to find a service item by ID

};

exports.updateServiceItem = async (req, res) => {

// your code to update a service item

};

exports.deleteServiceItem = async (req, res) => {

try {

const serviceItem = await ServiceItem.findByIdAndDelete(req.params.id);

if (!serviceItem) {

console.log(`Service item not found with id: ${req.params.id}`);

return res.status(404).render('error', { message: 'ServiceItem not found' });

}

console.log(`Service item deleted: ${serviceItem.name}`);

res.redirect('/items'); // Redirect back to the items page after deletion

} catch (error) {

console.error(`Error deleting service item: ${error.message}`, error);

res.status(400).render('error', { message: error.message });

}

};

exports.renderServiceItemsPage = async (req, res) => {

// your code to render the manage service items page

};

```

### Improvements Summary

1. \*\*CSRF Tokens\*\*: Properly included.

2. \*\*Error Handling\*\*: Ensure consistency.

3. \*\*Controllers\*\*: Use controllers for logic.

4. \*\*Model Imports\*\*: Ensure models are correctly imported in controllers.

5. \*\*Routes\*\*: Clarify and correct HTTP methods in the router.

**settingsRoutes.js**

Overall, the code appears functional and well-written, but there are a few areas where improvements could be made to enhance readability, performance, and security. Below are some specific points of interest, followed by a revised version of the code incorporating these suggestions:

### Code Review and Suggestions

1. \*\*`bcrypt` is not required\*\*:

- You are using `bcrypt` for password comparison and hashing, but it is not imported or required at the beginning of the file.

2. \*\*Input Validation\*\*:

- Consider validating the input (e.g., user Id, passwords, email) to ensure they meet the required format and constraints.

- Use a library like `Joi` or `express-validator`.

3. \*\*Error Handling\*\*:

- Enhance error handling to provide more specific error messages.

- Use custom error classes for different error scenarios (e.g., `UserNotFoundError`, `ValidationError`, etc.).

4. \*\*Asynchronous Operations\*\*:

- The current implementation uses try-catch for async operations, which is good practice. Make sure all async routes follow this pattern.

5. \*\*Session and CSRF Handling\*\*:

- Ensure the session and CSRF handling is robust and secure. CSRF tokens are already implemented, which is great.

6. \*\*Middleware Order\*\*:

- Middleware like `isAuthenticated` and `csrfProtection` should be consistently placed before route handler functions.

7. \*\*Miscellaneous\*\*:

- Logger messages should not expose sensitive information.

- Clean up and clear the response object to avoid sending unnecessary data.

### Revised Code

Here is the improved version of your original code incorporating some of the suggestions above:

```javascript

const express = require('express');

const router = express.Router();

const multer = require('multer');

const path = require('path');

const csrf = require('csurf');

const bcrypt = require('bcrypt');

const isAuthenticated = require('../middleware/authMiddleware').isAuthenticated;

const User = require('../models/User');

const logger = require('../logger');

// Setup CSRF protection middleware

const csrfProtection = csrf({ cookie: true });

// Setup multer for file upload

const storage = multer.diskStorage({

destination: function (req, file, cb) {

cb(null, 'uploads/');

},

filename: function (req, file, cb) {

cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));

}

});

const fileFilter = (req, file, cb) => {

if (file.mimetype === 'image/jpeg' || file.mimetype === 'image/png') {

cb(null, true);

} else {

cb(new Error('Unsupported file format'), false);

}

};

const upload = multer({

storage: storage,

limits: {

fileSize: 1024 \* 1024 \* 5 // 5MB max file size

},

fileFilter: fileFilter

});

// POST route to handle password change

router.post('/change-password', isAuthenticated, csrfProtection, async (req, res) => {

const { currentPassword, newPassword, confirmNewPassword } = req.body;

const userId = req.session.userId;

if (newPassword !== confirmNewPassword) {

return res.status(400).json({ success: false, message: 'New passwords do not match.' });

}

try {

const user = await User.findById(userId);

if (!user) {

return res.status(404).json({ success: false, message: 'User not found' });

}

const isMatch = await bcrypt.compare(currentPassword, user.password);

if (!isMatch) {

return res.status(400).json({ success: false, message: 'Current password is incorrect.' });

}

const hashedPassword = await bcrypt.hash(newPassword, 10);

user.password = hashedPassword;

await user.save();

logger.info(`Password updated successfully for userId: ${userId}`, {

method: "POST",

path: "/settings/change-password",

sessionId: req.sessionID,

timestamp: new Date().toISOString()

});

res.json({ success: true, message: 'Password updated successfully.' });

} catch (error) {

logger.error(`Error updating password: ${error.message}`, {

error: error,

method: "POST",

path: "/settings/change-password",

sessionId: req.sessionID,

timestamp: new Date().toISOString()

});

res.status(500).json({ success: false, message: 'Error updating password.', error: error.message });

}

});

// GET settings page with authentication and CSRF protection

router.get('/', isAuthenticated, csrfProtection, async (req, res) => {

try {

const userId = req.session.userId;

const user = await User.findById(userId);

if (!user) {

return res.status(404).send('User not found');

}

res.render('settings', {

title: 'Settings',

csrfToken: req.csrfToken(),

user: user // Pass the user object to the template

});

} catch (error) {

logger.error(`Error fetching user information: ${error.message}`, {

error: error,

method: "GET",

path: "/settings",

sessionId: req.sessionID,

timestamp: new Date().toISOString()

});

res.status(500).send('Error fetching user information.');

}

});

// POST route to handle form submission from the 'User Information' tab with CSRF protection

router.post('/', isAuthenticated, csrfProtection, upload.single('businessLogo'), async (req, res) => {

const { businessName, name, phoneNumber, email, streetAddress, city, state, zipCode, websiteURL, socialMediaLinks } = req.body;

let hoursOfOperation = {};

['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday', 'sunday'].forEach(day => {

hoursOfOperation[day] = {

from: req.body[`${day}From`],

to: req.body[`${day}To`]

};

});

try {

const userId = req.session.userId;

const user = await User.findById(userId);

if (!user) {

return res.status(404).json({ success: false, message: 'User not found.' });

}

user.businessName = businessName;

if (req.file) {

user.businessLogo = req.file.path; // Save file path if file was uploaded

}

user.name = name;

user.phoneNumber = phoneNumber;

user.email = email;

user.streetAddress = streetAddress;

user.city = city;

user.state = state;

user.zipCode = zipCode;

user.hoursOfOperation = hoursOfOperation;

user.websiteURL = websiteURL;

user.socialMediaLinks = socialMediaLinks;

await user.save();

logger.info(`User information updated successfully for userId: ${userId}`, {

method: "POST",

path: "/settings",

sessionId: req.sessionID,

timestamp: new Date().toISOString()

});

res.json({ success: true, message: 'User information updated successfully.', csrfToken: req.csrfToken() });

} catch (error) {

logger.error(`Error saving user information: ${error}`, {

error: error,

method: "POST",

path: "/settings",

sessionId: req.sessionID,

timestamp: new Date().toISOString()

});

res.status(500).json({

success: false,

message: 'Error saving user information.',

error: error.message

});

}

});

module.exports = router;

```

### Summary of Changes:

1. Imported `bcrypt` which was missing.

2. Improved error handling with specific responses.

3. Made some error responses more specific, including user not found errors.

4. Ensured that the middleware (`isAuthenticated`, `csrfProtection`) was placed correctly before the route handlers.

5. Simplified and cleaned up the code for clarity.

These changes should help make your code more maintainable, secure, and user-friendly.

**AuthMiddleware.js**

Your code in `authMiddleware.js` looks relatively solid. However, here are a few suggestions and improvements you might want to consider:

1. \*\*Improve Error Message Consistency\*\*:

Rather than sending a plain text response when the user is not authenticated, consider sending a JSON response. This approach can be more consistent and informative, especially for RESTful APIs.

2. \*\*Logging for Debugging\*\*:

Add some logging to help with debugging and to provide more visibility into what's happening within your middleware.

3. \*\*Session Checking Enhancement\*\*:

Depending on your session management library, you might consider verifying the session's validity in more detail. Sometimes library-specific hooks or methods are available for checking session integrity.

4. \*\*Code Readability and Comments\*\*:

Adding comments for clarity can be helpful for other developers who might be reading your code.

Here’s an improved version of your `authMiddleware.js`:

```javascript

const isAuthenticated = (req, res, next) => {

// Check if session exists and if it contains the userId

if (req.session && req.session.userId) {

console.log(`User authenticated: ${req.session.userId}`); // Logging for debug purposes

return next(); // User is authenticated, proceed to the next middleware/route handler

} else {

console.warn('User not authenticated'); // Log a warning for unauthenticated access attempts

return res.status(401).json({error: 'You are not authenticated'}); // Send a JSON response with a meaningful error message

}

};

module.exports = {

isAuthenticated

};

```

### Breakdown:

1. \*\*Logging\*\*:

```javascript

console.log(`User authenticated: ${req.session.userId}`);

console.warn('User not authenticated');

```

Logger statements help to trace the execution flow.

2. \*\*Consistent Error Messaging\*\*:

```javascript

return res.status(401).json({error: 'You are not authenticated'});

```

Using JSON format helps maintain consistency in API responses.

3. \*\*Comments\*\*:

Each logical block is commented to enhance code readability.

4. \*\*Security Considerations\*\*:

Make sure you are not inadvertently exposing any sensitive information logs or responses. Consider redacting sensitive information based on your security requirements.

These changes can improve the clarity, maintainability, security, and consistency of your `isAuthenticated` middleware.

**Crmintegration.js**

Your JavaScript (Express.js) code for CRM integration is generally well-structured and functional. However, there are certain improvements and suggestions that can enhance its robustness, readability, and maintainability:

### Improvements and Suggestions:

1. \*\*Consistent Logging\*\*:

- Avoid redundant error logging. Currently, the `logError` function logs the error and then another console.error statement is used.

2. \*\*Configuration Validation\*\*:

- Perform environment variable validation once rather than in each function call. This reduces redundancy and ensures validation is always up-to-date on service startup.

3. \*\*Error Handling\*\*:

- When re-throwing errors, it might be useful to wrap them into a custom error class to preserve context.

4. \*\*General Code Cleanup\*\*:

- Remove redundant logging and ensure consistent error handling practices.

5. \*\*Additional Error Information\*\*:

- Include HTTP status codes and response data in error logs for better debugging.

Here's your refactored code incorporating the aforementioned suggestions:

```javascript

const axios = require('axios');

const dotenv = require('dotenv');

dotenv.config();

const log = (message) => console.log(`[CRM Integration] ${message}`);

const logError = (error, message) => console.error(`[CRM Integration] ${message}: ${error.message}`, error.stack);

const validateEnvVars = () => {

const requiredVars = ['CRM\_API\_URL', 'CRM\_API\_KEY'];

const missingVars = requiredVars.filter(varName => !process.env[varName]);

if (missingVars.length > 0) {

throw new Error(`Missing required environment variables for CRM integration: ${missingVars.join(', ')}`);

}

};

// Validate environment variables once during startup

validateEnvVars();

const sendClientDataToCRM = async (clientData) => {

try {

const response = await axios.post(`${process.env.CRM\_API\_URL}/clients`, clientData, {

headers: {

'Authorization': `Bearer ${process.env.CRM\_API\_KEY}`,

'Content-Type': 'application/json',

},

});

log(`Client data sent to CRM successfully. Client ID: ${response.data.id}`);

return response.data;

} catch (error) {

logError(error, 'Failed to send client data to CRM.');

if (error.response) {

console.error(`HTTP Status: ${error.response.status}\nResponse Data: ${JSON.stringify(error.response.data)}`);

}

throw error;

}

};

const sendQuoteDataToCRM = async (quoteData) => {

try {

const response = await axios.post(`${process.env.CRM\_API\_URL}/quotes`, quoteData, {

headers: {

'Authorization': `Bearer ${process.env.CRM\_API\_KEY}`,

'Content-Type': 'application/json',

},

});

log(`Quote data sent to CRM successfully. Quote ID: ${response.data.id}`);

return response.data;

} catch (error) {

logError(error, 'Failed to send quote data to CRM.');

if (error.response) {

console.error(`HTTP Status: ${error.response.status}\nResponse Data: ${JSON.stringify(error.response.data)}`);

}

throw error;

}

};

module.exports = {

sendClientDataToCRM,

sendQuoteDataToCRM,

};

```

### Explanation of Changes:

1. \*\*Validation on Startup\*\*:

- The environment variables are validated at the start rather than inside each function to avoid redundancy.

2. \*\*Consolidated Logging\*\*:

- Removed unnecessary logging and consolidated error information, including HTTP status codes and response data, to aid in debugging.

3. \*\*Return Values\*\*:

- The API response data is returned after a successful API call to allow the caller to use the result as needed.

These improvements will make the code more maintainable and easier to debug in case of issues.

**distanceCalculator.js**

Your JavaScript code for calculating distances using the Google Distance Matrix API looks well-written and functional. Here are some specific comments, suggestions, and improvements:

### 1. Dependency and Environment Variable

- Your code correctly ensures that `axios` is required and the `GOOGLE\_API\_KEY` environment variable is set. This is good practice.

### 2. Input Validation

- The input validation function (`validateInput`) is simple and effective.

### 3. API Call and Error Handling

- Your code makes an API call using `axios` and properly checks for common errors. Good use of `try-catch` to handle potential errors.

### Improvements and Suggestions

#### 1. Detailed Response Status Check

- Sometimes the `response.data.rows[0].elements[0].status` might not be `OK` even if `response.data.status` is `OK`. We should check for this as well.

```javascript

if (response.data.status === 'OK' && response.data.rows[0].elements[0].status === 'OK') {

// proceed normally

}

```

#### 2. Improving Error Messages

- Provide detailed error messages to make debugging easier and give more context.

```javascript

console.error(`Error calculating distance: ${error.message}. Response: ${JSON.stringify(response.data)}`);

```

#### 3. Logging

- Too many logs might clutter your console. You could replace `console.log` with more sophisticated logging solutions if you use it in a production environment.

#### 4. Optional Chaining and Nullish Coalescing

- In the distance extraction, you can use optional chaining (`?.`) and nullish coalescing (`??`) to make the code more robust.

```javascript

const distanceInMeters = response.data?.rows?.[0]?.elements?.[0]?.distance?.value ?? null;

if (distanceInMeters) {

const distanceInMiles = distanceInMeters / 1609.34;

console.log(`Distance calculated: ${distanceInMiles} miles`);

return distanceInMiles;

} else {

console.error(`Distance information missing in response.`);

return null;

}

```

#### 5. Returning Detailed Errors

- Instead of returning `null` upon errors, consider returning detailed error information to the caller so it can handle errors appropriately.

```javascript

return { success: false, message: error.message, error };

```

#### 6. Use of Constants

- Use constants for the units conversion factor to improve readability.

```javascript

const METERS\_TO\_MILES = 1609.34;

const distanceInMiles = distanceInMeters / METERS\_TO\_MILES;

```

### Updated `calculateDistance` Function

Here's a more refined version incorporating the suggestions above:

```javascript

const axios = require('axios');

if (!process.env.GOOGLE\_API\_KEY) {

throw new Error('GOOGLE\_API\_KEY is not set. Please set your Google API key in the environment variables.');

}

const { GOOGLE\_API\_KEY } = process.env;

function validateInput(origin, destination) {

if (typeof origin !== 'string' || typeof destination !== 'string' || origin.trim() === '' || destination.trim() === '') {

throw new Error('Origin and destination must be non-empty strings.');

}

}

const METERS\_TO\_MILES = 1609.34;

async function calculateDistance(origin, destination) {

validateInput(origin, destination);

try {

const response = await axios.get('https://maps.googleapis.com/maps/api/distancematrix/json', {

params: {

origins: origin,

destinations: destination,

key: GOOGLE\_API\_KEY,

units: 'imperial'

}

});

if (response.data.status === 'OK' && response.data.rows[0].elements[0].status === 'OK') {

const distanceInMeters = response.data.rows[0].elements[0].distance.value;

const distanceInMiles = distanceInMeters / METERS\_TO\_MILES;

console.log(`Distance calculated: ${distanceInMiles} miles`);

return { success: true, distance: distanceInMiles };

} else {

console.error(`Google Distance Matrix API error: ${response.data.status}, Element status: ${response.data.rows[0].elements[0].status}`);

return { success: false, message: `API error: ${response.data.status}` };

}

} catch (error) {

console.error(`Error calculating distance: ${error.message}`, error);

return { success: false, message: error.message, error };

}

}

module.exports = { calculateDistance };

```

This refined version provides better error handling, detailed logging, and maintains readability.

**emailSender.js**

Your code is overall quite good, but there are a few areas for improvement and some best practices you might want to consider:

### 1. Type Conversion for PORT:

Since `process.env.EMAIL\_PORT` will be a string, you should convert it to a number if your email service expects it as such.

### 2. Secure Connection:

The `secure` field is hardcoded to `false`. It might be better to configure it based on environment variables as well.

### 3. Default Values:

Consider providing default values for environment variables for better validation and flexibility.

### 4. Error Handling:

When rethrowing an error, consider whether you want to throw a new error with different or more user-friendly messaging information, or simply rethrow the existing one.

### 5. Testing for Non-production:

For non-production environments (like development), you might want to mock or use a service like Mailtrap instead of actually sending emails.

### 6. Use of Constants:

Store constant/commonly used strings as constants.

### 7. Code Comments and Documentation:

More comprehensive commenting and documentation will help maintainability.

Here’s a refactored version of your code incorporating these improvements:

```javascript

const nodemailer = require('nodemailer');

const dotenv = require('dotenv');

dotenv.config();

const REQUIRED\_CONFIGS = ['EMAIL\_HOST', 'EMAIL\_PORT', 'EMAIL\_USER', 'EMAIL\_PASS'];

const DEFAULT\_EMAIL\_PORT = 587; // Default port for non-SSL email

// Validate environment variables for email configuration

const validateEmailConfig = () => {

const missingConfigs = REQUIRED\_CONFIGS.filter(config => !process.env[config]);

if (missingConfigs.length > 0) {

throw new Error(`Missing required email configuration environment variables: ${missingConfigs.join(', ')}`);

}

};

validateEmailConfig();

// Convert port to a number and check if secure connection should be used

const emailPort = Number(process.env.EMAIL\_PORT) || DEFAULT\_EMAIL\_PORT;

const useSecureConnection = emailPort === 465; // Typically 465 is true for secure

// Create a reusable transporter object using the default SMTP transport

const transporter = nodemailer.createTransport({

host: process.env.EMAIL\_HOST,

port: emailPort,

secure: useSecureConnection, // true for 465, false for other ports

auth: {

user: process.env.EMAIL\_USER,

pass: process.env.EMAIL\_PASS,

},

});

// Send email function

const sendEmail = async (emailOptions) => {

try {

// emailOptions should include, at a minimum, "to", "subject", and "text" or "html"

let info = await transporter.sendMail(emailOptions);

console.log(`Email sent: ${info.messageId}`);

} catch (error) {

console.error(`Error sending email: ${error.message}`, error);

throw new Error(`Failed to send email. Please try again later.`); // Providing a user-friendly error message

}

};

module.exports = { sendEmail };

```

### Explanation of Changes:

1. \*\*Type Conversion for Port\*\*:

- Converted `process.env.EMAIL\_PORT` to a number with a fallback default value.

2. \*\*Secure Connection\*\*:

- Automatically determine whether to use a secure connection based on the port number.

3. \*\*Constants and Default Values\*\*:

- Moved strings and default values to constants to make the code more readable and maintainable.

4. \*\*Error Handling\*\*:

- Provided user-friendly error messages when rethrowing errors.

5. \*\*Environment-Specific Setup\*\*:

- Consider adding checks for different environments (like development, testing, production) if applicable, e.g., to use mock transporters in development.

By making these changes, your code becomes more robust, maintainable, and aligned with best practices.

**pricingCalculator.js**

The provided code is a pricing calculator for service items using JavaScript and Express.js (or Node.js). Let's analyze the code for any errors, improvements, and suggestions:

### Errors:

No significant errors have been identified in the given code. The logic appears sound, and the use of `try-catch` blocks for error handling is good practice.

### Improvements & Suggestions:

1. \*\*Code Duplication\*\*

- The function `calculateBundledPrice` essentially calls `calculateItemizedPrice` and then modifies the `breakdown`. This results in some code duplication. You might simplify it without having the extra error handling because `calculateItemizedPrice` already handles errors.

2. \*\*Descriptive Error Messages\*\*

- While the error messages are already quite informative, you might want to include more context, such as which function these errors were caught in.

3. \*\*Tax Rate Handling\*\*

- If the `taxRateInput` is `undefined`, the code defaults it to 0. You might want to add checks for other invalid values like negative numbers.

4. \*\*Destructuring for Readability\*\*

- Instead of referencing `quote.serviceItems` multiple times, you could use destructuring.

5. \*\*Returning Consistent Data Structures\*\*

- The `calculateBundledPrice` returns a breakdown with a single item 'Bundled Service', which is consistent, but it uses a different structure than the `calculateItemizedPrice` function. This is generally fine but ensure the consuming code can handle these differences seamlessly.

### Code Improvement Example:

Here's the improved version incorporating the suggestions:

```javascript

const ServiceItem = require('../models/serviceItem');

const calculateItemizedPrice = async (serviceItems, taxRateInput = 0) => {

let subtotal = 0;

let breakdown = [];

for (const item of serviceItems) {

try {

const serviceItem = await ServiceItem.findById(item.serviceItemId);

if (!serviceItem) throw new Error(`Service item not found with ID: ${item.serviceItemId}`);

const price = item.customPrice !== undefined ? item.customPrice : serviceItem.price;

const itemTotal = price \* item.quantity;

subtotal += itemTotal;

breakdown.push({

name: serviceItem.name,

quantity: item.quantity,

price: price,

total: itemTotal

});

} catch (error) {

console.error(`Error calculating itemized price for service item with ID: ${item.serviceItemId}: ${error.message}`, error);

throw error;

}

}

// Validate taxRateInput

const taxRate = typeof taxRateInput === 'number' && taxRateInput >= 0 ? taxRateInput : 0;

const taxAmount = subtotal \* taxRate;

const totalPrice = subtotal + taxAmount;

return { subtotal, taxAmount, totalPrice, breakdown };

};

const calculateBundledPrice = async (serviceItems, taxRateInput) => {

const { subtotal, taxAmount, totalPrice } = await calculateItemizedPrice(serviceItems, taxRateInput);

return { subtotal, taxAmount, totalPrice, breakdown: [{ name: 'Bundled Service', total: totalPrice }] };

};

const calculateQuotePrice = async (quote, option = 'itemized', taxRateInput) => {

const { serviceItems } = quote;

switch (option) {

case 'bundled':

return calculateBundledPrice(serviceItems, taxRateInput);

case 'itemized':

default:

return calculateItemizedPrice(serviceItems, taxRateInput);

}

};

module.exports = { calculateQuotePrice };

```

### Summary:

This code review mainly focuses on readability, consistency, and minor code improvements that ensure robustness. The error handling in place is good, and the core logic does not suffer from major issues. The suggested changes help reduce redundancy and maintain clarity.

**quotePricingCalculator.js**

The provided JavaScript code defines a function `calculatePrice` that calculates the price of a service based on various parameters such as service type, frequency, rooms, square footage, extras, and distance. Overall, the code appears functional, but there are several areas where it can be improved for better readability, maintainability, and performance.

### Key Points for Improvement

1. \*\*Type Checking and Validations\*\*: Ensure that inputs are valid and appropriately typed to prevent runtime errors.

2. \*\*Separation of Concerns\*\*: Consider breaking the function into smaller functions to modularize components like mileage calculation, frequency discount, etc.

3. \*\*Error Handling\*\*: Adding error handling could make the code more robust.

4. \*\*Code Comments and Documentation\*\*: Adding more comments and possibly JavaDoc-style documentation will help other developers understand the code better.

5. \*\*Performance Improvements\*\*: Avoid unnecessary calculations and iterations.

### Improved Code

```javascript

const calculatePrice = (serviceType, frequency, rooms, squareFootage, extras, distance) => {

console.log(`Calculating price for serviceType: ${serviceType}, frequency: ${frequency}, squareFootage: ${squareFootage}, distance: ${distance}`);

// Constants

const pricePerSquareFoot = 0.01;

const mileageCharge = 0.55;

const mileageThreshold = 20;

// Input Validations

if (typeof squareFootage !== 'number' || squareFootage < 0) {

throw new Error("SquareFootage should be a positive number");

}

if (typeof distance !== 'number' || distance < 0) {

throw new Error("Distance should be a positive number");

}

let basePrice = squareFootage \* pricePerSquareFoot;

let extraCharges = calculateExtras(extras);

basePrice += calculateRoomCharges(rooms);

if (distance > mileageThreshold) {

const extraMileage = distance - mileageThreshold;

extraCharges += extraMileage \* mileageCharge;

}

basePrice = applyFrequencyDiscount(basePrice, serviceType, frequency);

const totalPrice = basePrice + extraCharges;

console.log(`Total calculated price: $${totalPrice.toFixed(2)}`);

return totalPrice;

};

const calculateExtras = (extras) => {

if (!Array.isArray(extras)) {

throw new Error("Extras should be an array");

}

return extras.reduce((total, extra) => {

if (typeof extra.price !== 'number' || extra.price < 0) {

throw new Error(`Invalid extra price: ${extra.price}`);

}

console.log(`Adding extra charge for: ${extra}`);

return total + extra.price;

}, 0);

};

const calculateRoomCharges = (rooms) => {

if (!Array.isArray(rooms)) {

throw new Error("Rooms should be an array");

}

return rooms.reduce((total, room) => {

if (typeof room.price !== 'number' || room.price < 0) {

throw new Error(`Invalid room price: ${room.price}`);

}

console.log(`Adding charge for room: ${room.name}`);

return total + room.price;

}, 0);

};

const applyFrequencyDiscount = (basePrice, serviceType, frequency) => {

if (serviceType === 'Recurring') {

console.log(`Adjusting price for recurring service with frequency: ${frequency}`);

switch (frequency) {

case 'Weekly':

return basePrice \* 0.9; // 10% discount for weekly

case 'Bi-Weekly':

return basePrice \* 0.95; // 5% discount for bi-weekly

case 'Monthly':

return basePrice \* 0.97; // 3% discount for monthly

default:

console.log(`No frequency discount applied.`);

return basePrice;

}

}

return basePrice;

};

module.exports = {

calculatePrice

};

```

### Summary of Improvements

1. \*\*Input Validation\*\*: Added validation for negative numbers for `squareFootage` and `distance`.

2. \*\*Modular Functions\*\*: Introduced helper functions (`calculateExtras`, `calculateRoomCharges`, and `applyFrequencyDiscount`) to break down the logic into manageable parts.

3. \*\*Error Handling\*\*: Included error handling for invalid data formats and types.

4. \*\*Code Comments\*\*: Improved with comments and better logging for debugging.

5. \*\*Performance\*\*: Used `reduce` for iteration and calculation, a more functional approach.

These improvements make the code cleaner, more maintainable, and easier to understand and test.

**clientDetails.ejs**

This code seems to be an EJS template for displaying client details on a web page. After a careful review, here are some errors identified, along with suggested improvements:

### Errors

1. \*\*Duplicate `<title>` Tag:\*\*

- There are two `<title>` tags in the `<head>` section.

```html

<title>Create Item</title>

<title>Client Details</title>

```

2. \*\*Broken Link in "Create Quote for Client":\*\*

- There is an extra slash in the link that should be removed.

```html

<a href="clients//<%= client.\_id %>/new" class="btn btn-success mt-3">Create Quote for Client</a>

```

3. \*\*Unclosed `<footer>` Tag:\*\*

- The `<footer>` tag is not properly closed.

```html

<footer class="footer fixed-bottom bg-light">

<div class="container text-center my-2">

<span>Copyright &copy; <%= 1900 + new Date().getYear() %> cleaning-tools</span>

</div>

<!-- Missing closing slash --></footer>

```

### Improvements and Suggestions

1. \*\*Layout Consistency:\*\*

- Ensure the layout is consistent and well-structured by avoiding repetitive `<title>` tags.

- Corrected:

```html

<head>

<meta charset="UTF-8">

<title>Client Details</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

</head>

```

2. \*\*Correct the Broken Link:\*\*

```html

<a href="/clients/<%= client.\_id %>/new" class="btn btn-success mt-3">Create Quote for Client</a>

```

3. \*\*Fix the Unclosed Footer Tag:\*\*

```html

<footer class="footer fixed-bottom bg-light">

<div class="container text-center my-2">

<span>&copy; <%= 1900 + new Date().getYear() %> cleaning-tools</span>

</div>

</footer>

```

4. \*\*Remove Unnecessary Comments:\*\*

- Comments like `<!-- Dynamic client details -->` don't add much value and could be removed for cleaner code:

```html

<div id="clientDetails" class="mt-3">

<p>Name: <span id="clientName"><%= client.name %></span></p>

<p>Email: <span id="clientEmail"><%= client.email %></span></p>

<p>Phone Number: <span id="clientPhoneNumber"><%= client.phoneNumber %></span></p>

<p>Client ID: <span id="clientId"><%= client.\_id %></span></p>

</div>

```

5. \*\*Bootstrap 5 Navbar Toggler:\*\*

- Ensure Bootstrap 5 classes are correctly applied. `data-bs-toggle` and `data-bs-target` should be used instead of `data-toggle` and `data-target`.

```html

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav">

<span class="navbar-toggler-icon"></span>

</button>

```

### Final Revised Version:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Client Details</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link" href="/">Home</a></li>

<li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>

<li class="nav-item"><a class="nav-link" href="/quotes">Create Quote</a></li>

<li class="nav-item"><a class="nav-link" href="/clients">Clients</a></li>

<li class="nav-item"><a class="nav-link" href="/items">Items</a></li>

<li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>

<% if (session && session.userId) { %>

<li class="nav-item"><a class="nav-link" href="/auth/logout">Logout</a></li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

</ul>

</div>

</nav>

<div class="container mt-5">

<h1>Client Details</h1>

<div id="clientDetails" class="mt-3">

<p>Name: <span id="clientName"><%= client.name %></span></p>

<p>Email: <span id="clientEmail"><%= client.email %></span></p>

<p>Phone Number: <span id="clientPhoneNumber"><%= client.phoneNumber %></span></p>

<p>Client ID: <span id="clientId"><%= client.\_id %></span></p>

</div>

<a href="/clients/<%= client.\_id %>/edit" class="btn btn-primary mt-3">Edit Client</a>

<a href="/clients/<%= client.\_id %>/new" class="btn btn-success mt-3">Create Quote for Client</a>

</div>

<footer class="footer fixed-bottom bg-light">

<div class="container text-center my-2">

<span>&copy; <%= 1900 + new Date().getYear() %> cleaning-tools</span>

</div>

</footer>

<script src="/js/main.js"></script>

</body>

</html>

```

These corrections and improvements should make the EJS template more robust and coherent. Happy coding!

**Clients.ejs**

Upon analyzing the provided `clients.ejs` file, here are the observations, potential errors, suggestions, and improvements:

### Issues to Address

1. \*\*Bootstrap JavaScript and Popper Integration\*\*:

Bootstrap 5 requires Popper.js for menu toggling in the navbar, which your script import seems to miss. Including Popper.js is necessary for full Bootstrap functionality.

2. \*\*Conflicting Data Attribute and Bootstrap Version\*\*:

Your Navbar uses `data-toggle` for the button, which is from Bootstrap 4. In Bootstrap 5, this should be `data-bs-toggle`.

3. \*\*CSRF Token Injection\*\*:

Confirm that the CSRF token is properly passed to the view to prevent CSRF attacks.

4. \*\*Session Check\*\*:

Ensure the `session` object is properly passed to the EJS template from the server-side to execute the conditional rendering correctly.

### Suggestions and Improvements

1. \*\*Use Bootstrap 5 Data Attributes\*\*:

Update `data-toggle` and `data-target` attributes to Bootstrap 5's `data-bs-toggle` and `data-bs-target`.

2. \*\*Ensure Proper Script Loading Order\*\*:

Load Popper.js as suggested by Bootstrap documentation, ensuring Bootstrap scripts work correctly.

3. \*\*Improve Accessibility\*\*:

Add `aria-label` to improve accessibility where appropriate, for example, on the navbar toggler button.

4. \*\*Handling Empty Client List\*\*:

Provide a user-friendly message or UI element if the `clients` array is empty to improve user experience.

5. \*\*Styling and Responsiveness\*\*:

Ensure the footer does not overlap with content when the page content is short, by using a sticky footer approach if necessary.

### Updated Code Example

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Cleaning Tools</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link" href="/">Home</a></li>

<li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>

<li class="nav-item"><a class="nav-link" href="/quotes">Create Quote</a></li>

<li class="nav-item active"><a class="nav-link" href="/clients">Clients</a></li>

<li class="nav-item"><a class="nav-link" href="/items">Items</a></li>

<li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>

<% if (session && session.userId) { %>

<li class="nav-item"><a class="nav-link" href="/auth/logout">Logout</a></li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

</ul>

</div>

</nav>

<div class="container mt-5">

<h1>Clients</h1>

<div class="mb-3">

<label for="clientSearch" class="form-label">Search Clients</label>

<input type="text" class="form-control" id="clientSearch" placeholder="Search for a client...">

<div id="clientResults" class="list-group mt-2"></div>

</div>

<% if (clients.length === 0) { %>

<p>No clients found.</p>

<% } else { %>

<table class="table table-striped">

<thead>

<tr>

<th>Name</th>

<th>Email</th>

<th>Phone Number</th>

<th>Actions</th>

</tr>

</thead>

<tbody>

<% clients.forEach(client => { %>

<tr>

<td><%= client.name %></td>

<td><%= client.email %></td>

<td><%= client.phoneNumber %></td>

<td>

<a href="/clients/<%= client.\_id %>/edit" class="btn btn-warning">Edit</a>

<form action="/clients/<%= client.\_id %>?\_method=DELETE" method="POST" style="display:inline;">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<button type="submit" class="btn btn-danger" onclick="return confirm('Are you sure you want to delete this client?');">Delete</button>

</form>

</td>

</tr>

<% }) %>

</tbody>

</table>

<% } %>

<a href="/clients/new" class="btn btn-primary">Add New Client</a>

</div>

<footer class="footer bg-light mt-auto">

<div class="container text-center my-2">

<span>&copy; <%= 1900 + new Date().getYear() %> Cleaning Tools</span>

</div>

</footer>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.5/dist/umd/popper.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.min.js" integrity="sha384-BBtl+eGJRgqQAUMxJ7pMwbEyER4l1g+O15P+16Ep7Q9Q+zqX6gSbd85u4mG4QzX+" crossorigin="anonymous"></script>

<script src="/js/clientSearch.js"></script>

<script src="/js/clientDeleteConfirmation.js"></script>

</body>

</html>

```

This revised code addresses the identified issues and includes improvements for better compatibility, accessibility, and user experience.

**createClient.ejs**

Your EJS template appears to be well-structured and functional. Here are some suggestions for improvements and considerations for errors:

### Error Checking

1. \*\*CSRF Token\*\*: Ensure that the `csrfToken` is correctly provided by the server-side code to prevent CSRF attacks.

```ejs

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

```

### Improvements

1. \*\*Form Validation and Enhancements\*\*:

- Add `required` attribute to the `phoneNumber` if it's mandatory.

- Use input types and patterns for `phoneNumber`, `zip` to ensure correct formats.

```html

<div class="mb-3">

<label for="phoneNumber" class="form-label">Phone Number</label>

<input type="tel" class="form-control" id="phoneNumber" name="phoneNumber" pattern="[0-9]{10}" title="Enter a valid phone number">

</div>

<div class="mb-3">

<label for="zip" class="form-label">Zip</label>

<input type="text" class="form-control" id="zip" name="zip" pattern="[0-9]{5}" title="Enter a valid zip code">

</div>

```

2. \*\*Accessibility\*\*: Ensure all elements are accessible.

- Add `for` attributes to labels matching the `id` of their corresponding input fields.

- Provide helpful tooltips or hints using `aria-\*` attributes.

3. \*\*Styling\*\*: Add more Bootstrap classes if needed for better design, or define custom classes in your CSS files.

- You can add `col` classes for a grid layout.

- Add classes for margins and paddings like `mb-3`, `mt-3`, etc.

- Consider using Bootstrap input groups if needed.

4. \*\*Error Handling\*\*: Display error messages if form submission fails due to server-side validation.

```ejs

<% if (typeof errors !== 'undefined') { %>

<div class="alert alert-danger">

<ul>

<% errors.forEach(function(error) { %>

<li><%= error.msg %></li>

<% }); %>

</ul>

</div>

<% } %>

```

5. \*\*Security\*\*: Make sure to sanitize all inputs on the server-side to prevent XSS and SQL injection attacks.

6. \*\*Enhancements\*\*:

- Use `autocomplete` attribute to improve user experience.

```html

<input type="text" class="form-control" id="name" name="name" required autocomplete="name">

```

- Consider providing a dropdown for `state` to minimize errors in input.

```html

<label for="state" class="form-label">State</label>

<select class="form-control" id="state" name="state">

<option value="">Select State</option>

<option value="CA">California</option>

<option value="NY">New York</option>

<!-- Add all state options -->

</select>

```

### Final Revised Code

Here is an example incorporating some of these suggestions:

```ejs

<!DOCTYPE html>

<html lang="en">

<head>

<%- include('partials/\_head.ejs') %>

<title>Create New Client</title>

</head>

<body>

<%- include('partials/\_header.ejs') %>

<div class="container mt-5">

<h1>Create New Client</h1>

<% if (typeof errors !== 'undefined') { %>

<div class="alert alert-danger">

<ul>

<% errors.forEach(function(error) { %>

<li><%= error.msg %></li>

<% }); %>

</ul>

</div>

<% } %>

<form action="/clients" method="POST">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="mb-3">

<label for="name" class="form-label">Name</label>

<input type="text" class="form-control" id="name" name="name" required autocomplete="name">

</div>

<div class="mb-3">

<label for="email" class="form-label">Email</label>

<input type="email" class="form-control" id="email" name="email" required autocomplete="email">

</div>

<div class="mb-3">

<label for="phoneNumber" class="form-label">Phone Number</label>

<input type="tel" class="form-control" id="phoneNumber" name="phoneNumber" pattern="[0-9]{10}" title="Enter a valid phone number" autocomplete="tel">

</div>

<div class="mb-3">

<label for="streetAddress" class="form-label">Street Address</label>

<input type="text" class="form-control" id="streetAddress" name="streetAddress" autocomplete="street-address">

</div>

<div class="mb-3">

<label for="city" class="form-label">City</label>

<input type="text" class="form-control" id="city" name="city" autocomplete="address-level2">

</div>

<div class="mb-3">

<label for="state" class="form-label">State</label>

<select class="form-control" id="state" name="state">

<option value="">Select State</option>

<option value="CA">California</option>

<option value="NY">New York</option>

<!-- Add all state options -->

</select>

</div>

<div class="mb-3">

<label for="zip" class="form-label">Zip</label>

<input type="text" class="form-control" id="zip" name="zip" pattern="[0-9]{5}" title="Enter a valid zip code" autocomplete="postal-code">

</div>

<button type="submit" class="btn btn-primary">Create Client</button>

</form>

</div>

<%- include('partials/\_footer.ejs') %>

<script src="/js/main.js"></script>

</body>

</html>

```

By implementing these improvements, you enhance the maintainability, usability, and security of your form.

**createItem.ejs**

### Analysis and Suggestions for `createItem.ejs`

This code is generally well-structured and follows good practices. Below, I offer an analysis of the code and provide suggestions to improve it further:

#### 1. \*\*Meta tags and Bootstrap\*\*

- Ensure you have all necessary meta tags for responsive design.

- Update Bootstrap script to use the correct integrity and crossorigin attributes for security.

#### 2. \*\*Navbar Togglers\*\*

- `data-toggle` and `data-target` should be updated as follows to be compatible with Bootstrap 5:

```html

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

```

#### 3. \*\*Session Handling\*\*

- Ensure the `session` variable is correctly passed from the server to EJS.

- Consider adding session-based user data to the navbar (e.g., displaying a user's name).

```html

<% if (session && session.userId) { %>

<li class="nav-item"><a class="nav-link" href="/auth/logout">Logout (<%= session.userName %>)</a></li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

```

#### 4. \*\*Form Elements\*\*

- Use proper autocomplete values (`off`, `on`, or specific field names).

- Consider client-side validation and user guidance.

```html

<form action="/items" method="POST">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="mb-3">

<label for="name" class="form-label">Item Name</label>

<input type="text" class="form-control" id="name" name="name" required autocomplete="name">

</div>

<div class="mb-3">

<label for="description" class="form-label">Description</label>

<input type="text" class="form-control" id="description" name="description" autocomplete="description">

</div>

<div class="mb-3">

<label for="categorySelect" class="form-label">Category</label>

<select class="form-control" id="categorySelect" name="category" required>

<option value="" disabled selected>Select category</option>

<option value="Standard">Standard</option>

<option value="Extras">Extras</option>

<option value="Package">Package</option>

<option value="Service">Service</option>

</select>

</div>

<div class="mb-3">

<label for="unitsSelect" class="form-label">Units</label>

<select class="form-control" id="unitsSelect" name="units" required>

<option value="" disabled selected>Select units type</option>

<option value="Per Item">Per Item</option>

<option value="Per Hour">Per Hour</option>

</select>

</div>

<div class="mb-3">

<label for="price" class="form-label">Price</label>

<input type="number" class="form-control" id="price" name="price" required step="0.01" min="0" autocomplete="off">

</div>

<button type="submit" class="btn btn-primary">Save</button>

</form>

```

#### 5. \*\*Footer\*\*

- The footer is fixed to the bottom. Be sure this is the intended behavior.

- If it's dynamic, consider using CSS to prevent overlap with content.

#### 6. \*\*JavaScript Libraries\*\*

- For Bootstrap, ensure you include the correct version and integrity hashes.

- Ensure JavaScript files are correctly loaded after including optional CDN files.

```html

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

<script src="/js/main.js"></script>

```

#### 7. \*\*Security\*\*

- Ensure `\_csrf` is properly generated and passed.

- Use HTTPS for all connections and verify sources for CDN.

- Sanitize inputs on the server-side to prevent injection attacks.

### Summary

- \*\*Meta tags\*\*: Ensure all necessary viewport settings are included.

- \*\*Bootstrap\*\*: Update `data-\*` attributes for Bootstrap 5 compatibility.

- \*\*Session Handling\*\*: Ensure session data is passed and utilized effectively.

- \*\*Forms\*\*: Enable proper autocomplete attributes and consider front-end validation.

- \*\*Footer\*\*: Confirm the desired behavior of a fixed footer.

- \*\*JavaScript Libraries\*\*: Ensure the correct versions and integrity hashes.

- \*\*Security\*\*: Continue to validate and sanitize inputs both client and server-side.

By implementing these improvements and suggestions, you'll enhance the functionality, security, and user experience of your web page.

**~~createQuote.ejs~~**

The provided EJS template and accompanying JavaScript code appear quite comprehensive, however, there are several potential improvements and suggestions for better practice and error-proofing the code:

### General Suggestions:

1. \*\*Semantic HTML\*\*: Consider using appropriate HTML elements to improve accessibility and semantics.

2. \*\*Code Repetition\*\*: There are multiple places, especially within the JavaScript code, where similar logic is repeated. Extracting these into functions would make the code more maintainable.

3. \*\*Async/Await\*\*: Ensure error handling in async functions covers possible failure scenarios comprehensively.

### Structural and Semantic Improvements:

1. \*\*Unique IDs\*\*: When adding attachments and contracts, the `id` attributes are duplicated for multiple file inputs. Each `input` element should have a unique ID.

```javascript

<label for="attachments">Upload Attachments</label>

<input type="file" id="attachments" name="attachments" class="form-control" multiple>

```

Instead:

```javascript

const uniqueId = 'unique-id-' + (new Date()).getTime(); // Ensure ID is unique

<label for="${uniqueId}">Upload Attachments</label>

<input type="file" id="${uniqueId}" name="attachments" class="form-control" multiple>

```

2. \*\*Modal Form IDs\*\*: Pay attention to IDs within modals. Both `createClientForm` and `editClientForm` use the same `id` attributes for some fields, which might cause issues since multiple forms reside in the same DOM context.

```html

<input type="text" class="form-control" id="clientName" name="name" autocomplete="name">

<input type="text" class="form-control" id="editClientName" name="name" autocomplete="name">

```

### JavaScript Improvements:

1. \*\*Dynamic Attachment and Contract IDs\*\*:

Use a unique identifier for new `input` fields when adding Attachments or Contracts to avoid conflicts.

```javascript

document.getElementById('addAttachmentBtn').addEventListener('click', () => {

const uniqueId = `attachments-${Date.now()}`;

// rest of the code

attachmentGroup.innerHTML = `

<label for="${uniqueId}">Upload Attachments</label>

<input type="file" id="${uniqueId}" name="attachments" class="form-control" multiple>

<button type="button" class="btn btn-danger mt-2 remove-attachment-btn">Remove</button>

`;

});

```

2. \*\*Error Handling in Axios Request\*\*:

More detailed error handling can give better feedback on why a request might fail.

```javascript

document.getElementById('calculateDistanceBtn').addEventListener('click', async () => {

const clientId = document.getElementById('clientId').value;

const userAddress = document.getElementById('userAddress').value;

const csrfToken = document.querySelector('input[name="\_csrf"]').value;

try {

const response = await axios.post('/quotes/calculate-distance', { clientId, userAddress }, {

headers: {

'CSRF-Token': csrfToken // Ensure CSRF token is sent in the header for validation

}

});

document.getElementById('distance').value = response.data.distance;

} catch (error) {

console.error('Error calculating distance:', error);

alert('Failed to calculate distance. Please try again.');

}

});

```

3. \*\*DRY Principle\*\*:

Extract repeated code into reusable functions, for instance, adding event listeners to remove buttons:

```javascript

function addRemoveButtonEventListeners(selector, containerSelector) {

document.querySelectorAll(selector).forEach(btn => {

btn.addEventListener('click', (event) => {

event.target.closest(containerSelector).remove();

});

});

}

addRemoveButtonEventListeners('.remove-attachment-btn', '.attachment-group');

addRemoveButtonEventListeners('.remove-contract-btn', '.contract-group');

```

### Form Handling:

1. \*\*Form Validation\*\*:

Ensure the forms have proper validation both client-side and server-side. The `required` attribute is used, but JavaScript validation can supplement it.

```javascript

document.getElementById('quoteForm').addEventListener('submit', (event) => {

const fields = ['clientId', 'title', 'scopeOfWork'];

let isValid = true;

fields.forEach((field) => {

const input = document.getElementById(field);

if (!input.value.trim()) {

isValid = false;

input.classList.add('is-invalid');

} else {

input.classList.remove('is-invalid');

input.classList.add('is-valid');

}

});

if (!isValid) {

event.preventDefault();

alert('Please fill in all required fields.');

}

});

```

### EJS and HTML Structural Suggestions:

1. \*\*Dynamic CSS Libraries\*\*:

Consider adding versioning or using a more reliable method to serve CSS libraries. Ensure Axios is loaded correctly.

```html

<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

```

### Accessibility:

1. \*\*ARIA Attributes\*\*:

Add ARIA attributes for better accessibility on interactive elements like buttons and modals.

```html

<button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>

```

Adding `aria-describedby`, `aria-labelledby`, where necessary, can significantly improve accessibility.

By applying these suggestions, the provided code should be more maintainable, error-free, and user-friendly.

**~~editClient.ejs~~**

The provided code is well-written and covers most of the essentials for an Express.js and EJS-based edit client page. Below are some potential errors, improvements, and suggestions for better practices:

### Errors

1. \*\*Double `<title>` Element\*\*:

- There are two `<title>` tags, which is incorrect. Remove one of them:

```html

<title>Cleaning Tools</title>

```

2. \*\*Bootstrap JavaScript Library\*\*:

- You're missing the Bootstrap JavaScript library which is essential for the navbar toggling to work. Add this script at the end of the body:

```html

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

```

3. \*\*CSRF Token Handling\*\*:

- Ensure your server is actually passing a `csrfToken`. This is necessary for CSRF protection.

4. \*\*Meta Tags for Responsiveness\*\*:

- Add a viewport meta tag for better responsiveness:

```html

<meta name="viewport" content="width=device-width, initial-scale=1">

```

### Improvements and Suggestions

1. \*\*Client-Side Validation\*\*:

- Add better client-side validations using HTML5 attributes:

```html

<input type="email" class="form-control" id="email" name="email" value="<%= client.email %>" required pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,}$">

```

2. \*\*Bootstrap Modal for Success Message\*\*:

- Instead of using `alert`, you can use a Bootstrap modal for a more modern approach.

```html

<!-- Modal -->

<div class="modal fade" id="updateModal" tabindex="-1" aria-labelledby="updateModalLabel" aria-hidden="true">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title" id="updateModalLabel">Update Status</h5>

<button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>

</div>

<div class="modal-body">

Client updated successfully!

</div>

<div class="modal-footer">

<button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>

</div>

</div>

</div>

</div>

<script>

const urlParams = new URLSearchParams(window.location.search);

if (urlParams.has('updated')) {

const myModal = new bootstrap.Modal(document.getElementById('updateModal'));

myModal.show();

}

</script>

```

3. \*\*Icon for Nav Items\*\*:

- Adding icons to navigation items can enhance the UI.

4. \*\*Avoid Inline JavaScript\*\*:

- Prefer loading JavaScript in a separate file for cleaner code and better maintenance.

5. \*\*Enhanced Security\*\*:

- Add `Content-Security-Policy` and other security-related headers in your Express app.

6. \*\*Accessible Footer\*\*:

- Make the footer more accessible and not fixed-bottom if it’s not necessary. If you need it to stick to the bottom, ensure there’s enough content to push it there naturally.

```html

<footer class="footer bg-light">

<div class="container text-center my-2">

<span>&copy; <%= new Date().getFullYear() %> cleaning-tools</span>

</div>

</footer>

```

```css

/\* CSS for ensuring content doesn't go under footer \*/

html {

position: relative;

min-height: 100%;

}

body {

margin-bottom: 60px;

}

.footer {

position: absolute;

bottom: 0;

width: 100%;

height: 60px; /\* Set the fixed height of the footer here \*/

line-height: 60px; /\* Vertically center the text there \*/

}

```

### Final Note

Ensure that these suggestions align with your overall application architecture and coding standards. The goal is to enhance the UX/UI experience and maintain a secure and clean codebase.

**~~editItem.ejs~~**

The provided EJS template is mostly well-structured and seems functional. However, there are a few areas that can be improved upon or corrected for better performance, security, and readability. Here's a detailed analysis:

### Errors and Improvements

1. \*\*Inconsistencies in HTML and Bootstrap Version Usage\*\*:

- The data-toggle attribute is not necessary for modern versions of Bootstrap.

- Missing jQuery Script: Bootstrap 5 does not use jQuery so the data-toggle no longer holds.

```diff

- <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav">

+ <button class="navbar-toggler" type="button" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

```

2. \*\*CSRF Protection\*\*:

- Ensure that `csrfToken` is being properly passed from the server to the frontend.

3. \*\*Optional Fields\*\*:

- Ensure that optional fields (like description) are set correctly.

```diff

+ <input type="text" class="form-control" id="description" name="description" value="<%= item.description %>">

```

4. \*\*Category Select Default Option\*\*:

- Set a default option that indicates a user to choose instead of "Category:".

```diff

- <option selected>Category:</option>

+ <option value="" disabled selected>Select a category</option>

```

5. \*\*CSRF Token Handling\*\*:

- verify `\_csrf` token handling is necessary.

6. \*\*Footer Current Year\*\*:

- Using `new Date().getFullYear()` is a simpler approach.

```diff

- <span>Copyright &copy; <%= 1900 + new Date().getYear() %> cleaning-tools</span>

+ <span>&copy;<%= new Date().getFullYear() %> cleaning-tools</span>

```

7. \*\*Form Submission Method Override\*\*:

- Ensure the correct setup of `\_method` for PUT requests. Ensure the backend respects it.

### Code with Applied Improvements:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Cleaning Tools</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link" href="/">Home</a></li>

<li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>

<li class="nav-item"><a class="nav-link" href="/quotes">Create Quote</a></li>

<li class="nav-item"><a class="nav-link" href="/clients">Clients</a></li>

<li class="nav-item"><a class="nav-link" href="/items">Items</a></li>

<li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>

<% if (session && session.userId) { %>

<li class="nav-item"><a class="nav-link" href="/auth/logout">Logout</a></li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

</ul>

</div>

</nav>

<div class="container mt-5">

<h1>Edit Item</h1>

<form action="/items/<%= item.\_id %>?\_method=PUT" method="POST">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="mb-3">

<label for="name" class="form-label">Item Name</label>

<input type="text" class="form-control" id="name" name="name" value="<%= item.name %>" required>

</div>

<div class="mb-3">

<label for="description" class="form-label">Description</label>

<input type="text" class="form-control" id="description" name="description" value="<%= item.description %>">

</div>

<div class="mb-3" id="categoryDiv">

<label for="category" class="form-label">Category</label>

<select class="form-control" id="categorySelect" name="category" required>

<option value="" disabled selected>Select a category</option>

<option value="Standard" <%= item.category === 'Standard' ? 'selected' : '' %>>Standard</option>

<option value="Extras" <%= item.category === 'Extras' ? 'selected' : '' %>>Extras</option>

<option value="Package" <%= item.category === 'Package' ? 'selected' : '' %>>Package</option>

<option value="Service" <%= item.category === 'Service' ? 'selected' : '' %>>Service</option>

</select>

</div>

<div class="mb-3">

<label for="price" class="form-label">Price</label>

<input type="text" class="form-control" id="price" name="price" value="<%= item.price %>" required>

</div>

<button type="submit" class="btn btn-primary">Save</button>

</form>

</div>

<footer class="footer fixed-bottom bg-light">

<div class="container text-center my-2">

<span>&copy; <%= new Date().getFullYear() %> cleaning-tools</span>

</div>

</footer>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-BBtl+eGJRgqQAUMxJ7pMwbEyER4l1g+O15P+16Ep7Q9Q+zqX6gSbd85u4mG4QzX+" crossorigin="anonymous"></script>

<script src="/js/main.js"></script>

<script>

document.addEventListener('DOMContentLoaded', function() {

const urlParams = new URLSearchParams(window.location.search);

if (urlParams.has('success')) {

alert('Item saved successfully');

}

});

</script>

</body>

</html>

```

With these improvements, your form will be more secure and user-friendly. Additionally, all the Bootstrap elements will be up to date. Ensure that any changes made align with the server-side code to maintain functionality, especially around CSRF tokens, form handling, and authentication.

**~~editQuote.ejs~~**

The provided EJS file appears to be a well-structured form for editing a quote. Here's a thorough analysis covering potential errors, adjustments, and improvements:

### Errors and Fixes

1. \*\*Client Information Section\*\*

- \*\*Address label formatting\*\*: The closing `</b>` tag is misplaced in the "Address" label. It should be:

```html

<p><b>Address:</b> <%= quote.clientId.streetAddress %>, <%= quote.clientId.city %>, <%= quote.clientId.state %> <%= quote.clientId.zip %></p>

```

- \*\*Rendering issue\*\* for client information: Ensure `quote.clientId` object exists and contains the required fields.

2. \*\*JavaScript Errors\*\*

- \*\*User Address Element Existence\*\*: The button with ID `calculateDistanceBtn` relies on an element with ID `userAddress`. Ensure `userAddress` exists on the form or handle the case where it might not exist to prevent JavaScript errors.

- \*\*Multiple File Inputs\*\*: In the attachments and contracts sections, the `name` attribute of the file input should include `[]` to allow multiple file uploads correctly. Example:

```html

<input type="file" id="attachments<%= index %>" name="attachments[]" class="form-control" multiple>

```

### Recommendations and Improvements

1. \*\*Button Responsiveness\*\*

- After adding or removing attachments/contracts, ensure the index is consistently managed to avoid potential conflicts in element IDs.

2. \*\*CSRF Token Handling\*\*

- Ensure that your CSRF token's implementation is secure and properly updated in your back-end logic.

3. \*\*Form Validation and User Feedback\*\*

- Implement basic form validation feedback to the user. Use JavaScript to provide real-time feedback for required fields that are not filled out.

- Example:

```javascript

document.getElementById('quoteForm').addEventListener('submit', function(event) {

const requiredFields = document.querySelectorAll('[required]');

let isValid = true;

requiredFields.forEach(field => {

if (!field.value) {

field.classList.add('is-invalid');

isValid = false;

} else {

field.classList.remove('is-invalid');

}

});

if (!isValid) {

event.preventDefault(); // Prevent submit if invalid

}

});

```

4. \*\*Dynamic Sections Toggle\*\*

- For dynamic sections like Frequency and Initial Cleaning Options based on the selected service type, use JavaScript to show or hide elements correctly:

```javascript

document.getElementById('serviceType').addEventListener('change', function() {

const frequencyDiv = document.getElementById('frequencyDiv');

const initialCleaningDiv = document.getElementById('initialCleaningDiv');

if (this.value === 'Recurring') {

frequencyDiv.classList.remove('d-none');

initialCleaningDiv.classList.add('d-none');

} else if (this.value === 'One-Time Deep Clean') {

frequencyDiv.classList.add('d-none');

initialCleaningDiv.classList.remove('d-none');

} else {

frequencyDiv.classList.add('d-none');

initialCleaningDiv.classList.add('d-none');

}

});

```

5. \*\*Attachment and Contract Removal\*\*

- To ensure a smooth user experience and prevent potential memory leaks or event listener conflicts, manage event listeners efficiently.

```javascript

document.addEventListener('click', function(event) {

if (event.target.classList.contains('remove-attachment-btn')) {

event.target.closest('.attachment-group').remove();

}

if (event.target.classList.contains('remove-contract-btn')) {

event.target.closest('.contract-group').remove();

}

});

```

6. \*\*Global Event Listeners\*\*

- Instead of adding event listeners individually, use event delegation for more efficient event handling, especially for dynamically added elements.

### Additional Suggestions

- \*\*Improved UX with Visual Cues\*\*: Give users visual confirmation when adding new line items, attachments, or contracts.

```javascript

function addElement(container, element) {

container.appendChild(element);

element.scrollIntoView({ behavior: 'smooth' });

element.classList.add('highlight');

setTimeout(() => element.classList.remove('highlight'), 1000);

}

```

By making these adjustments and improvements, the form will not only function correctly but also provide a more seamless and intuitive user experience.

**~~Error.js~~**

Overall, your EJS template looks well-structured and correctly formatted to render an error page. However, considering best practices and potential improvements, here are several suggestions:

### Code Review:

1. \*\*HTML Structure\*\*:

- The HTML structure is correct, containing essential meta tags and linking to stylesheets.

2. \*\*Bootstrap CDN\*\*:

- The Bootstrap CDN link is valid and correctly included.

3. \*\*Custom CSS\*\*:

- The custom stylesheet `/css/style.css` is linked correctly, assuming that the static files are being served properly in your Express application.

### Suggestions and Improvements:

1. \*\*Meta Tags for Responsiveness\*\*:

- Adding a viewport meta tag helps ensure the page is properly responsive on various devices.

```html

<meta name="viewport" content="width=device-width, initial-scale=1">

```

Place this tag inside your `<head>` section:

```html

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>Error</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

</head>

```

2. \*\*Error Handling and Default Message\*\*:

- Ensure that `message` is always defined when rendering the error page. If it's possible that `message` might not be passed, consider providing a default message to display.

```ejs

<p><%= message || 'Something went wrong. Please try again later.' %></p>

```

3. \*\*Semantic HTML\*\*:

- Wrap the main content inside a `<main>` tag for a better semantic structure:

```html

<body>

<div class="container mt-5">

<main>

<h1 class="text-danger">Error</h1>

<p><%= message || 'Something went wrong. Please try again later.' %></p>

<a href="/items" class="btn btn-primary">Back to Items</a>

</main>

</div>

</body>

```

4. \*\*Accessibility Improvements\*\*:

- Ensure headings and links are more accessible, e.g., by improving focus styles and valid ARIA attributes if necessary.

```html

<a href="/items" class="btn btn-primary" aria-label="Back to Items">Back to Items</a>

```

5. \*\*Link Security\*\*:

- When linking to external stylesheets or scripts in production, make sure to use `integrity` and `crossorigin` attributes for added security.

```html

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" integrity="sha384...YourIntegrityHash..." crossorigin="anonymous">

```

6. \*\*Static Assets Serving\*\*:

- Ensure that your Express.js application is serving static files correctly. Typically this is done with:

```javascript

app.use(express.static(path.join(\_\_dirname, 'public')));

```

Ensure your CSS file is in a correct path, for example `public/css/style.css`.

### Final Code Example:

Putting it all together, your final `error.ejs` file could look like this:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>Error</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" integrity="sha384...YourIntegrityHash..." crossorigin="anonymous">

<link rel="stylesheet" href="/css/style.css">

</head>

<body>

<div class="container mt-5">

<main>

<h1 class="text-danger">Error</h1>

<p><%= message || 'Something went wrong. Please try again later.' %></p>

<a href="/items" class="btn btn-primary" aria-label="Back to Items">Back to Items</a>

</main>

</div>

</body>

</html>

```

By incorporating these improvements, your `error.ejs` template will be more robust, responsive, and accessible.

**~~Index.ejs~~**

The provided code is an EJS template for an Express-based web application. Overall, the code structure is clean and adheres to common practices. However, there are some potential improvements and considerations to note:

1. \*\*Error Handling:\*\*

- Ensure `<%= csrfToken %>` is correctly passed to the view. If `csrfToken` is not defined, it can lead to undefined values in the form, which could cause the CSRF protection to fail.

2. \*\*SEO and Accessibility Improvements:\*\*

- Ensure the `lang` attribute on the `<html>` tag matches the language of your site content.

- Add `<title>` and `<meta>` tags within the included `\_head.ejs` partial for SEO and accessibility improvements.

- Consider adding a `<meta charset="UTF-8">` tag to the head to ensure correct character encoding.

3. \*\*Stylistic Improvements:\*\*

- Indentation and spacing are generally good, but it’s always good to maintain consistency.

- Use semantic HTML elements where possible for better structure and readability.

4. \*\*Security Considerations:\*\*

- Ensure all included partials (`\_head.ejs`, `\_header.ejs`, and `\_footer.ejs`) are sanitizing any user data to prevent XSS attacks.

- Ensure you are using `Helmet.js` or similar middleware to secure HTTP headers.

5. \*\*Form Structure:\*\*

- Ensure that you have necessary `<label>` tags for form fields for accessibility.

- Make sure that other form fields (besides CSRF) are included when the form is used in a real scenario.

6. \*\*Client-Side JavaScript:\*\*

- Confirm that `/js/main.js` is loaded properly and doesn’t contain any errors.

Here's a suggestion incorporating some of the points mentioned above:

```ejs

<!DOCTYPE html>

<html lang="en">

<%- include('partials/\_head.ejs') %> <!-- Make sure this partial includes meta tags and title -->

<body>

<%- include('partials/\_header.ejs') %>

<main role="main" class="container mt-4">

<div class="text-center">

<h1>Cleaning Tools</h1>

</div>

<!-- Sample form to demonstrate CSRF token usage -->

<form action="/some-endpoint" method="POST">

<input type="hidden" name="\_csrf" value="<%= csrfToken || '' %>"> <!-- Default to an empty string if csrfToken is not defined -->

<!-- Make sure to add label elements and other form fields -->

<label for="exampleField">Example Field:</label>

<input type="text" id="exampleField" name="exampleField" required>

<button type="submit">Submit</button>

</form>

</main>

<%- include('partials/\_footer.ejs') %>

<script src="/js/main.js"></script> <!-- Ensure this is working correctly -->

</body>

</html>

```

\*\*Example `\_head.ejs` Partial:\*\*

```ejs

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta name="description" content="Description of this page">

<title>Cleaning Tools</title>

<!-- Add any other meta tags or link tags for stylesheets -->

</head>

```

By ensuring these best practices, your code will be more robust, secure, and maintainable.

**~~Items.ejs~~**

Your EJS template for rendering the service items page looks solid overall, but there are a few areas where we can improve and ensure correctness:

### 1. Meta Tags & Accessibility

Consider adding some meta tags for better SEO and accessibility:

```html

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta name="description" content="Manage your cleaning tools service items efficiently.">

```

### 2. CSRF Token Handling

Your script is trying to send a CSRF token in the headers, but there's no input element for `\_csrf` in the HTML. You might need to add a hidden input field for CSRF token if you're using a CSRF protection middleware in your Express app:

```html

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

```

Make sure your backend renders the `csrfToken` variable in the template data.

### 3. Code Improvements

- \*\*JavaScript Strict Mode\*\*: Wrap your JavaScript in a `strict` context.

- \*\*Error-Handling and Asynchronous Functions\*\*: Consider using async/await for better readability and handling fetch errors more clearly.

- \*\*Event Delegation\*\*: Instead of adding event listeners to each delete button, leverage event delegation for better performance.

### 4. Bootstrap Correctness

For Bootstrap components (such as navbars), ensure you're using the correct class names and attributes.

### Improved Code (With Suggested Changes):

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta name="description" content="Manage your cleaning tools service items efficiently.">

<title>Cleaning Tools</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link" href="/">Home</a></li>

<li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>

<li class="nav-item"><a class="nav-link" href="/quotes">Create Quote</a></li>

<li class="nav-item"><a class="nav-link" href="/clients">Clients</a></li>

<li class="nav-item"><a class="nav-link" href="/items">Items</a></li>

<li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>

<% if (session && session.userId) { %>

<li class="nav-item"><a class="nav-link" href="/auth/logout">Logout</a></li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

</ul>

</div>

</nav>

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="container mt-5">

<h1>Manage Service Items</h1>

<div class="mb-3">

<a href="/items/new" class="btn btn-primary">Create New Item</a>

</div>

<% if (serviceItems.length > 0) { %>

<table class="table table-bordered">

<thead>

<tr>

<th>Item Name</th>

<th>Description</th>

<th>Category</th>

<th>Price</th>

<th>Actions</th>

</tr>

</thead>

<tbody id="serviceItemsTable">

<% serviceItems.forEach(function(item) { %>

<tr data-id="<%= item.\_id %>">

<td><%= item.name %></td>

<td><%= item.description %></td>

<td><%= item.category %></td>

<td>$<%= item.price.toFixed(2) %></td>

<td>

<a href="/items/<%= item.\_id %>/edit" class="btn btn-warning btn-sm">Edit</a>

<button class="btn btn-danger btn-sm delete-item-btn" data-id="<%= item.\_id %>">Delete</button>

</td>

</tr>

<% }); %>

</tbody>

</table>

<% } else { %>

<p>No service items found. <a href="/items/new">Create a new item</a>.</p>

<% } %>

</div>

<footer class="footer fixed-bottom bg-light">

<div class="container text-center my-2">

<span>&copy; <%= new Date().getFullYear() %> cleaning-tools</span>

</div>

</footer>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

<script>

'use strict';

document.addEventListener('DOMContentLoaded', () => {

const serviceItemsTable = document.getElementById('serviceItemsTable');

const csrfToken = document.querySelector('input[name="\_csrf"]').value;

serviceItemsTable.addEventListener('click', async (event) => {

if (event.target.classList.contains('delete-item-btn')) {

const row = event.target.closest('tr');

const itemId = event.target.getAttribute('data-id');

if (confirm('Are you sure you want to delete this item?')) {

try {

const response = await fetch(`/api/service-items/${itemId}`, {

method: 'DELETE',

headers: {

'Content-Type': 'application/json',

'CSRF-Token': csrfToken

}

});

const data = await response.json();

if (data.success) {

row.remove();

alert('Service item deleted successfully.');

} else {

alert('Error deleting service item: ' + data.error);

}

} catch (error) {

console.error('Error deleting service item:', error);

alert('Error deleting service item.');

}

}

}

});

});

</script>

</body>

</html>

```

### Summary of Changes:

1. Added the viewport meta tag for responsiveness.

2. Included a hidden input for the CSRF token.

3. Used `getFullYear()` instead of `getYear()`.

4. Used Bootstrap's data attribute for toggle.

5. Implemented event delegation in JavaScript for better performance.

6. Used modern async/await for clearer asynchronous code handling.

These improvements cover various aspects including better security handling, modern JavaScript practices, and improved performance.

**~~Login.ejs~~**

The provided EJS template for a login page is functional but contains some issues and areas that could benefit from improvements. Here’s an analysis touching on errors, improvements, and suggestions:

### Errors

1. \*\*Missing `<head>` Tag\*\*:

- The `<meta>`, `<title>` and `<link>` tags should be enclosed within a `<head>` tag.

2. \*\*Bootstrap Usage\*\*:

- The `data-toggle` and `data-target` attributes in the button need to be updated for Bootstrap 5.

### Improvements and Suggestions

1. \*\*Semantic HTML\*\*:

- Wrap `<meta charset="UTF-8">`, `<title>`, and `<link>` tags within a `<head>` tag for better semantics.

2. \*\*Bootstrap 5 Updates\*\*:

- Replace the `data-toggle` and `data-target` attributes with `data-bs-toggle` and `data-bs-target`, respectively.

- The `navbar-toggler-icon` should be wrapped inside a `<span>` tag.

3. \*\*Accessibility\*\*:

- Add `aria-label` attributes where necessary, especially for interactive elements for better accessibility.

- Ensure the text inside the \*\*login\*\* form is properly labeled with `<label>` tags.

4. \*\*Consistent CSRF Handling\*\*:

- Ensure that `csrfToken` is being correctly output in the form input field.

5. \*\*Script Placement\*\*:

- Place JavaScript includes at the end of the `<body>` tag just before closing the tag.

### Corrected Code

```ejs

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Cleaning Tools</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link" href="/">Home</a></li>

<li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>

<li class="nav-item"><a class="nav-link" href="/quotes">Create Quote</a></li>

<li class="nav-item"><a class="nav-link" href="/clients">Clients</a></li>

<li class="nav-item"><a class="nav-link" href="/items">Items</a></li>

<li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>

<% if (session && session.userId) { %>

<li class="nav-item"><a class="nav-link" href="/auth/logout">Logout</a></li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

</ul>

</div>

</nav>

<main role="main">

<div class="container mt-5">

<h2>Login</h2>

<form action="/auth/login" method="POST">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="mb-3">

<label for="username" class="form-label">Username</label>

<input type="text" id="username" name="username" placeholder="Username" required class="form-control">

</div>

<div class="mb-3">

<label for="password" class="form-label">Password</label>

<input type="password" id="password" name="password" placeholder="Password" required class="form-control">

</div>

<div class="mb-3">

<button type="submit" class="btn btn-primary">Login</button>

Don't have an account? <a href="/auth/register">Register</a>

</div>

</form>

</div>

</main>

<footer class="footer fixed-bottom bg-light">

<div class="container text-center my-2">

<span>Copyright &copy; <%= new Date().getFullYear() %> cleaning-tools</span>

</div>

</footer>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

</body>

</html>

```

Summary of Changes:

- Added a `<head>` section to correct semantic HTML structure.

- Updated `data-toggle` and `data-target` attributes to `data-bs-toggle` and `data-bs-target` for Bootstrap 5 compliance.

- Improved accessibility by adding `aria-controls`, `aria-expanded`, and `aria-label` attributes to the toggler button.

- Used `<label>` tags for form inputs to improve accessibility.

- Placed JavaScript includes just before the closing `</body>` tag for better page loading performance.

- Fixed the calculation of the year within the footer.

**~~quoteDetails.ejs~~**

Analyzing the given EJS (Embedded JavaScript) template file for displaying quote details, there are a few observations and suggestions for improvements:

### Errors

1. \*\*Incorrect HTML Tag Order in Address Line:\*\*

```html

<p><b>Address:</p></b>

```

The `</b>` tag should be inside the `<p>` tag:

```html

<p><b>Address:</b> <%= quote.clientId.streetAddress %>, <%= quote.clientId.city %>, <%= quote.clientId.state %> <%= quote.clientId.zip %></p>

```

### Improvements and Suggestions

1. \*\*Client Information Null Check:\*\*

Before trying to access nested properties, it's good practice to handle potential undefined values to avoid runtime errors. Use optional chaining in your EJS templates.

```html

<p><b>Client Name:</b> <%= quote.clientName %></p>

<p><b>Phone:</b> <%= quote.clientId?.phoneNumber ?? 'N/A' %></p>

<p><b>Email:</b> <%= quote.clientId?.email ?? 'N/A' %></p>

<p><b>Address:</b> <%= quote.clientId?.streetAddress ?? 'N/A' %>, <%= quote.clientId?.city ?? '' %>, <%= quote.clientId?.state ?? '' %> <%= quote.clientId?.zip ?? '' %></p>

```

2. \*\*Consistent Formatting:\*\*

Ensure a consistent file format by adding proper indentation and spacing in JS blocks.

```html

<% if (quote.scopeOfWork) { %>

<hr>

<p class="section-title">Scope of Work</p>

<p><%= quote.scopeOfWork %></p>

<% } %>

```

3. \*\*Accessibility Improvements:\*\*

Make sure your document is accessible by following best practices in using HTML elements.

- Add `alt` attributes to images.

- Ensure color contrast for readability.

4. \*\*Enhancing Security:\*\*

Use `helmet` or similar middleware in your Express.js server to set security-related HTTP headers and reduce certain kinds of attacks.

```javascript

const helmet = require('helmet');

app.use(helmet());

```

5. \*\*Substitute `null` Values:\*\*

Handle `null` values gracefully to provide fallback content.

```html

<p class="section-title">Client Information</p>

<p><b>Client Name:</b> <%= quote.clientName || 'N/A' %></p>

```

6. \*\*Avoid Double Calculation for Total Amount:\*\*

Instead of recalculating totals client-side, validate that pre-calculated values are correct server-side and pass them directly.

```html

<p>Subtotal: <span>$<%= quote.subtotal?.toFixed(2) || '0.00' %></span></p>

<p>Tax Rate: <span><%= quote.taxRate || '0' %>%</span></p>

<p>Total: <span>$<%= quote.total?.toFixed(2) || '0.00' %></span></p>

```

7. \*\*Script Tag Placement:\*\*

Move script tags to the end of the `body`. Minimize render-blocking resources.

```html

<body>

<!-- Your body content here -->

<script src="/js/clientSearch.js"></script>

</body>

```

8. \*\*CSS and JS Minification:\*\*

Use minified versions of your CSS and JavaScript files for faster loading times in production.

9. \*\*Error Handling for Attachments and Contracts:\*\*

Incorporate basic error handling and messaging if files are missing.

```html

<% if (quote.attachments && quote.attachments.length > 0) { %>

<hr>

<p class="section-title">Attachments</p>

<ul>

<% quote.attachments.forEach(function(attachment) { %>

<li>

<a href="/uploads/<%= attachment.savedFilename %>" target="\_blank"><%= attachment.originalFilename || 'Unnamed File' %></a>

</li>

<% }); %>

</ul>

<% } else { %>

<p>No attachments available.</p>

<% } %>

```

Make these adjustments to ensure a robust, consistent, and user-friendly application.

**~~Quotes.ejs~~**

The Express.js (EJS) code you posted for the `quotes.ejs` file is well-structured and functional. However, there are always areas for improvement and potential corrections. Below are some suggestions and notes to enhance the code:

### Error Checking and Fixes

1. \*\*CSRF Token Handling\*\*:

Ensure that `csrfToken` is correctly passed to the view. If not, it will cause the form submissions to fail:

```javascript

res.render('quotes', { quotes, csrfToken: req.csrfToken() });

```

2. \*\*\_method Override\*\*:

Double-check that your middleware for method override (`method-override` package) is correctly configured in your Express application to handle the `?\_method=DELETE`.

### Code Improvements and Enhancements

1. \*\*HTML Formatting and Semantic Tags\*\*:

Use semantic HTML5 tags (such as `<main>`, `<section>`, etc.) to improve the accessibility and readability of your HTML structure.

```html

<main class="container mt-5">

<!-- Rest of the code -->

</main>

```

2. \*\*Accessibility\*\*:

- Add `aria-label` or `aria-labelledby` to input fields and buttons to improve screen reader support.

```html

<input type="text" class="form-control" id="clientSearch" aria-label="Search for a client..." placeholder="Search for a client...">

```

3. \*\*Consistency in Button Styling\*\*:

Ensure all inline styling is moved to the CSS file for better maintainability.

```html

<form action="/quotes/<%= quote.\_id %>?\_method=DELETE" method="POST" class="d-inline">

```

4. \*\*Preventing Potential XSS\*\*:

Always escape any user-generated content to prevent XSS (Cross-Site Scripting).

```html

<td><%- quote.clientName %></td>

<td><%- quote.title %></td>

<td>$<%- quote.total.toFixed(2) %></td>

<td><%- quote.status %></td>

```

### Additional Suggestions:

1. \*\*Client-Side Search Functionality\*\*:

Ensure `clientSearch.js` effectively handles dynamic searching and the result population in `#clientResults`.

2. \*\*Mobile Responsiveness\*\*:

Ensure your page is mobile-friendly. Utilize Bootstrap's responsive classes appropriately.

3. \*\*Error Handling for Empty Quotes\*\*:

Add a message when no quotes are available.

```html

<tbody>

<% if (quotes.length === 0) { %>

<tr>

<td colspan="5" class="text-center">No quotes available</td>

</tr>

<% } else { %>

<% quotes.forEach(quote => { %>

<tr>

<td><%= quote.clientName %></td>

<td><%= quote.title %></td>

<td>$<%= quote.total.toFixed(2) %></td>

<td><%= quote.status %></td>

<td>

<a href="/quotes/<%= quote.\_id %>" class="btn btn-primary">View</a>

<a href="/quotes/<%= quote.\_id %>/edit" class="btn btn-warning">Edit</a>

<form action="/quotes/<%= quote.\_id %>?\_method=DELETE" method="POST" class="d-inline">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<button type="submit" class="btn btn-danger">Delete</button>

</form>

</td>

</tr>

<% }) %>

<% } %>

</tbody>

```

4. \*\*CSRF Token Meta Tag\*\*:

It's a good practice to include a CSRF token meta tag within the head so that it can be accessed by any JavaScript.

```html

<meta name="csrf-token" content="<%= csrfToken %>">

```

5. \*\*Organizing Scripts\*\*:

Ensure your client-side JavaScript is well-organized. You might consider adding event listeners directly in the `clientSearch.js` file rather than inline.

By addressing these points, your `quotes.ejs` template would be more secure, maintainable, readable, and user-friendly.

**Register.ejs**

The `register.ejs` file is well-structured and uses Bootstrap for styling, which ensures a responsive design. Here are some suggestions for improvements, error handling, and additional features:

### Improvements and Suggestions:

1. \*\*Input Validation for Phone Number and Zip Code:\*\*

- While you have set the `required` attribute, it’s also good to add basic pattern validation directly in the form elements.

```html

<input type="tel" pattern="[0-9]{10}" class="form-control" id="phoneNumber" name="phoneNumber" required>

<input type="text" pattern="[0-9]{5}" class="form-control" id="zipCode" name="zipCode" required>

```

2. \*\*Include `pattern` Attribute for Validation:\*\*

- Add `pattern` attributes to enforce basic input constraints on fields like username, phone number, and zip code.

3. \*\*CSRF Token Initialization Check:\*\*

- Ensure that the `csrfToken` is properly provided from your backend controller.

4. \*\*Consistent Naming Conventions:\*\*

- Ensure consistency in naming form elements. For example, consider naming `name` to `fullName` to avoid confusion with the attribute name.

5. \*\*Error Handling Messages:\*\*

- Output error messages provided by the server. Consider adding a section to display feedback if registration fails or succeeds.

```html

<% if (errorMessage) { %>

<div class="alert alert-danger"><%= errorMessage %></div>

<% } %>

<% if (successMessage) { %>

<div class="alert alert-success"><%= successMessage %></div>

<% } %>

```

6. \*\*Switch to `let` or `const` for Declaring Variables:\*\*

- Use `const` or `let` instead of `var` in your server-side scripts (although this is not applicable directly in this ejs file).

7. \*\*Accessibility Improvement:\*\*

- Ensure that accessibility is maintained. Consider adding `aria-label` attributes where necessary.

### Example Updated File:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Register</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

</head>

<body>

<div class="container mt-5">

<h2>Register</h2>

<% if (errorMessage) { %>

<div class="alert alert-danger"><%= errorMessage %></div>

<% } %>

<% if (successMessage) { %>

<div class="alert alert-success"><%= successMessage %></div>

<% } %>

<form action="/auth/register" method="POST">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="mb-3">

<label for="username" class="form-label">Username</label>

<input type="text" pattern="[a-zA-Z0-9]+" class="form-control" id="username" name="username" required>

</div>

<div class="mb-3">

<label for="password" class="form-label">Password</label>

<input type="password" class="form-control" id="password" name="password" required>

</div>

<div class="mb-3">

<label for="businessName" class="form-label">Business Name</label>

<input type="text" class="form-control" id="businessName" name="businessName" required>

</div>

<div class="mb-3">

<label for="name" class="form-label">Name</label>

<input type="text" class="form-control" id="name" name="name" required>

</div>

<div class="mb-3">

<label for="phoneNumber" class="form-label">Phone Number</label>

<input type="tel" pattern="[0-9]{10}" class="form-control" id="phoneNumber" name="phoneNumber" required>

</div>

<div class="mb-3">

<label for="email" class="form-label">Email</label>

<input type="email" class="form-control" id="email" name="email" required>

</div>

<div class="mb-3">

<label for="streetAddress" class="form-label">Street Address</label>

<input type="text" class="form-control" id="streetAddress" name="streetAddress" required>

</div>

<div class="mb-3">

<label for="city" class="form-label">City</label>

<input type="text" class="form-control" id="city" name="city" required>

</div>

<div class="mb-3">

<label for="state" class="form-label">State</label>

<input type="text" class="form-control" id="state" name="state" required>

</div>

<div class="mb-3">

<label for="zipCode" class="form-label">Zip Code</label>

<input type="text" pattern="[0-9]{5}" class="form-control" id="zipCode" name="zipCode" required>

</div>

<button type="submit" class="btn btn-primary">Register</button>

</form>

</div>

</body>

</html>

```

Make sure your server-side validation matches the client-side rules to ensure consistency and security. This revised code helps improve user experience, data integrity, and accessibility.

**Settings.ejs**

Your EJS template for a settings page using Bootstrap looks quite robust. However, there are a few potential issues and areas for improvement to consider:

### Potential Issues:

1. \*\*Bootstrap 5 Data Attribute Usage\*\*:

- You're using `data-toggle` and `data-target`, which is from Bootstrap 4, not Bootstrap 5. In Bootstrap 5, those should be `data-bs-toggle` and `data-bs-target`, respectively. This applies to your navbar toggler.

```html

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav">

<span class="navbar-toggler-icon"></span>

</button>

```

2. \*\*Logout Form Method\*\*:

- If the `/auth/logout` route is a POST route (typically logout actions are), you should use a form with a POST method instead of a simple link.

```html

<% if (session && session.userId) { %>

<li class="nav-item">

<form action="/auth/logout" method="POST">

<button type="submit" class="btn btn-link nav-link">Logout</button>

</form>

</li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

```

3. \*\*File Paths for Styles and Scripts\*\*:

- You are loading a script twice:

```html

<script src="/js/settings.js" defer></script>

...

<script src="/js/settings.js"></script>

```

Use only one instance:

```html

<script src="/js/settings.js" defer></script>

```

4. \*\*CSRF Tokens for Security\*\*:

- Ensure that CSRF tokens are correctly set and validated on the server side to prevent CSRF attacks.

### Improvements and Suggestions:

1. \*\*Form Validations\*\*:

- Add client-side form validation using HTML5 attributes or JavaScript for a better user experience. For example, phone number validation, password strength validation, etc.

2. \*\*Accessibility Improvements\*\*:

- Include `aria-` labels to improve accessibility, especially for interactive elements like the navbar toggle button.

```html

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

```

3. \*\*Styling Consistency and Design\*\*:

- Ensure that your design is fully responsive by testing on various devices.

- Use `col-md-6` or `col-lg-4` to better manage the layout on different screen sizes.

4. \*\*Error Handling and Feedback\*\*:

- Implement proper error handling and feedback mechanisms for the user, such as showing success messages upon saving settings or errors if something goes wrong.

5. \*\*Code Reuse and Cleanliness\*\*:

- Consider breaking down the form into partials if it gets larger, to maintain code cleanliness and reuse.

6. \*\*Security Best Practices\*\*:

- Always sanitize and validate user input both on the client side and server side.

- Ensure proper use of HTTPS to protect data in transit.

Here's a snippet adjusting some of these points:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Cleaning Tools</title>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/style.css">

<script src="/js/settings.js" defer></script>

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link" href="/">Home</a></li>

<li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>

<li class="nav-item"><a class="nav-link" href="/quotes">Create Quote</a></li>

<li class="nav-item"><a class="nav-link" href="/clients">Clients</a></li>

<li class="nav-item"><a class="nav-link" href="/items">Items</a></li>

<li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>

<% if (session && session.userId) { %>

<li class="nav-item">

<form action="/auth/logout" method="POST">

<button type="submit" class="btn btn-link nav-link">Logout</button>

</form>

</li>

<% } else { %>

<li class="nav-item"><a class="nav-link" href="/auth/login">Login</a></li>

<% } %>

</ul>

</div>

</nav>

...

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

</body>

</html>

```

Following these guidelines will ensure your application is more secure, user-friendly, accessible, and maintainable.

**\_footer.ejs**

The provided code is an EJS (Embedded JavaScript) template for rendering a footer in a web application. There are a few aspects to address for correctness and improvements.

### Error:

1. \*\*Incorrect Year Calculation\*\*:

The `new Date().getYear()` method does not return the current year directly. It returns the number of years since 1900. Therefore, `1900 + new Date().getYear()` should be replaced with `new Date().getFullYear()` to get the current year directly.

### Suggested Improvements:

1. \*\*Semantic HTML Elements\*\*:

- You might want to consider using more semantic HTML elements for better accessibility and readability, like `<small>` for smaller, less important content.

2. \*\*External Script Loading\*\*:

- Consider using the `defer` or `async` attribute while loading external scripts to improve page load performance by not blocking the rendering of the page. In this case, `defer` is more appropriate.

### Refactored Code:

Here is the refined version of your `\_footer.ejs`:

```ejs

<footer class="footer bg-light">

<div class="container text-center my-2">

<small>&copy; <%= new Date().getFullYear() %> cleaning-tools</small>

</div>

</footer>

<script defer src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>

```

### Explanation:

1. \*\*Year Calculation Fixed:\*\*

- Changed `1900 + new Date().getYear()` to `new Date().getFullYear()` to correctly display the current year.

2. \*\*Smaller Text for Copyright:\*\*

- Changed `<span>` to `<small>` for semantic HTML as copyright notices are typically less important.

3. \*\*Defer External Script Loading:\*\*

- Added the `defer` attribute to the script tag to ensure it doesn't block HTML parsing and improves the loading performance.

These changes ensure that the webpage footer correctly displays the current year, uses more semantic HTML, and loads JavaScript scripts more efficiently.

**\_head.ejs**

The provided EJS file contains the meta charset, title, and links to the stylesheet for a web page. Here's a detailed analysis, including possible errors, improvements, and suggestions:

### Analysis and Suggestions:

1. \*\*Meta Charset Tag\*\*:

- Current `<meta charset="UTF-8">` is correct and ensures that character encoding is set to UTF-8.

```html

<meta charset="UTF-8">

```

2. \*\*Title Tag\*\*:

- `<title>Cleaning Tools</title>` accurately reflects a simple and meaningful title.

```html

<title>Cleaning Tools</title>

```

- However, depending on the context of your website or application, you might want to consider adding a dynamic title.

```ejs

<title><%= pageTitle || 'Cleaning Tools' %></title>

```

This allows different pages to have unique titles if `pageTitle` is set.

3. \*\*External Stylesheets\*\*:

- The link to Bootstrap's stylesheet from a CDN is correct.

```html

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

```

- The custom stylesheet is also well-linked.

```html

<link rel="stylesheet" href="/css/style.css">

```

- Consider verifying the path to your custom stylesheet. Ensure that the `/css/style.css` file path is correct and accessible from the client-side.

### Possible Improvements:

1. \*\*Enhanced Performance with Subresource Integrity (SRI)\*\*:

- For better security and integrity, it’s good practice to include Subresource Integrity (SRI) attributes for the Bootstrap stylesheet. This helps ensure that the resource has not been tampered with.

```html

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" integrity="sha384-..." crossorigin="anonymous">

```

Make sure to replace `sha384-...` with the actual SRI hash from Bootstrap's CDN.

2. \*\*Preconnect and Defer Options for Performance\*\*:

- Using `<link rel="preconnect">` for the CDN can help improve the load times by making an early connection to the required domains.

```html

<link rel="preconnect" href="https://cdn.jsdelivr.net">

```

3. \*\*Compatibility Considerations\*\*:

- Include a `meta` tag for viewport settings, to ensure proper scaling on mobile devices.

```html

<meta name="viewport" content="width=device-width, initial-scale=1.0">

```

4. \*\*Comments for Maintainability\*\*:

- Adding comments can improve the maintainability of the file.

```html

<!-- Basic Meta Tags -->

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<!-- Title -->

<title>Cleaning Tools</title>

<!-- External Stylesheets -->

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" integrity="sha384-..." crossorigin="anonymous">

<link rel="stylesheet" href="/css/style.css">

```

### Final Suggested Code:

```ejs

<!-- Basic Meta Tags -->

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<!-- Title -->

<title><%= pageTitle || 'Cleaning Tools' %></title>

<!-- Performance and Security Enhancements -->

<link rel="preconnect" href="https://cdn.jsdelivr.net">

<!-- External Stylesheets -->

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" integrity="sha384-..." crossorigin="anonymous">

<link rel="stylesheet" href="/css/style.css">

```

By considering the above suggestions, you can improve the performance, security, and maintainability of your web page’s head section.

**Header.ejs**

The given EJS code represents a navigation bar typically found in an Express.js application using the Bootstrap framework. The code is generally well-structured, but there are some improvements and possible issues to consider:

1. \*\*Session Variable Usage\*\*:

- Ensure the `session` object is correctly passed to the EJS template from your Express.js route. Otherwise, this will cause reference errors.

2. \*\*Bootstrap Framework\*\*:

- Make sure that the Bootstrap CSS and JS files are properly included in your project, either through `<link>` tags in the HTML `<head>` or through a task runner/bundler.

3. \*\*Security Considerations\*\*:

- Perform server-side validation for the user session to prevent unauthorized access.

- Always validate and sanitize inputs on the server side.

4. \*\*Usability Improvements\*\*:

- Indicate the active page to provide better user experience by adding an `active` class to the respective navigation item. This can be done by conditionally checking the current path.

5. \*\*Accessibility\*\*:

- Ensure all `nav-item` elements have descriptive and distinguishable text for accessibility purposes.

- Consider adding `aria-expanded`, `aria-controls` attributes to the navbar toggler button to improve accessibility.

Here’s the revised code considering these aspects:

```ejs

<nav class="navbar navbar-expand-md navbar-dark bg-dark">

<a class="navbar-brand" href="/">Cleaning Tools</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item <%= (currentPath === '/' ? 'active' : '') %>">

<a class="nav-link" href="/">Home</a>

</li>

<li class="nav-item <%= (currentPath === '/dashboard' ? 'active' : '') %>">

<a class="nav-link" href="/dashboard">Dashboard</a>

</li>

<li class="nav-item <%= (currentPath === '/quotes' ? 'active' : '') %>">

<a class="nav-link" href="/quotes">Create Quote</a>

</li>

<li class="nav-item <%= (currentPath === '/clients' ? 'active' : '') %>">

<a class="nav-link" href="/clients">Clients</a>

</li>

<li class="nav-item <%= (currentPath === '/items' ? 'active' : '') %>">

<a class="nav-link" href="/items">Items</a>

</li>

<li class="nav-item <%= (currentPath === '/settings' ? 'active' : '') %>">

<a class="nav-link" href="/settings">Settings</a>

</li>

<% if (session && session.userId) { %>

<li class="nav-item">

<a class="nav-link" href="/auth/logout">Logout</a>

</li>

<% } else { %>

<li class="nav-item">

<a class="nav-link" href="/auth/login">Login</a>

</li>

<% } %>

</ul>

</div>

</nav>

```

### Key Changes and Suggestions:

1. Added `aria-controls`, `aria-expanded`, and `aria-label` attributes to the navbar-toggler button for better accessibility.

2. Included conditional logic to set the `active` class based on the `currentPath` variable to highlight the current page in the navigation menu.

3. Always make sure the session data is correctly passed to the template for the conditional rendering to work.

Be sure to define `currentPath` in your route logic so that it contains the current path being viewed, allowing the conditional `active` class assignment to work correctly. For example:

```javascript

app.get('/some-route', (req, res) => {

res.render('some-view', {

session: req.session,

currentPath: req.path,

});

});

```

This will ensure that the active link is highlighted according to the current page the user is on.