To use environment variables in Node.js applications, you can utilize the `process.env` object to access variables defined in your environment. Here's how you can modify the `connectMongoDb` function to use environment variables for the MongoDB connection string:

```javascript

const mongoose = require('mongoose');

require('dotenv').config(); // Load environment variables from .env file

const connectMongoDb = async () => {

try {

const connectionString = process.env.MONGODB\_URI; // Retrieve MongoDB URI from environment variables

await mongoose.connect(connectionString, {

useNewUrlParser: true,

useUnifiedTopology: true,

useCreateIndex: true,

});

console.log("Connected to MongoDB server via mongoose successfully");

} catch (error) {

console.error("Error connecting to MongoDB:", error.message);

process.exit(1);

}

};

module.exports = connectMongoDb;

```

In this code:

1. We first import the `dotenv` package to load environment variables from a `.env` file.

2. We define the MongoDB connection string using `process.env.MONGODB\_URI`, assuming you have a variable named `MONGODB\_URI` in your `.env` file containing the MongoDB connection details.

3. We use `await mongoose.connect(connectionString, { ... })` to connect to MongoDB using the retrieved connection string and the specified options.

4. Any errors during the connection attempt are caught and logged to the console.

Make sure to create a `.env` file in your project's root directory and add your MongoDB URI like this:

```

MONGODB\_URI=mongodb+srv://username:password@cluster-name.mongodb.net/database-name

```

Replace `username`, `password`, `cluster-name`, and `database-name` with your actual MongoDB credentials and database information.  
To create a `.env` file in your project's root directory, follow these steps:

1. Open a text editor or code editor of your choice.

2. Create a new file and name it `.env`. Note that the filename starts with a dot (`.`).

3. Add your environment variables to the `.env` file in the format `KEY=VALUE`. Each variable should be on a new line.

Here's an example of how a `.env` file might look like:

```

PORT=3000

MONGODB\_URI=mongodb://localhost/mydatabase

SECRET\_KEY=your\_secret\_key\_here

```

In this example:

- `PORT` is set to `3000`, which can be used as the port number for your server.

- `MONGODB\_URI` contains the MongoDB connection string.

- `SECRET\_KEY` is an example of a secret key that can be used for encryption or other purposes in your application.

After creating the `.env` file and adding your environment variables, you can access these variables in your Node.js application using the `dotenv` package. Here's an example of how to load and use environment variables from the `.env` file:

1. Install the `dotenv` package using npm or yarn:

```bash

npm install dotenv

# or

yarn add dotenv

```

2. In your Node.js application entry point (e.g., `index.js` or `app.js`), require and configure `dotenv` to load the environment variables from the `.env` file:

```javascript

require('dotenv').config();

```

3. Now you can access your environment variables using `process.env.VARIABLE\_NAME`. For example:

```javascript

const port = process.env.PORT || 3000; // Use PORT environment variable or default to 3000

const mongodbUri = process.env.MONGODB\_URI;

const secretKey = process.env.SECRET\_KEY;

console.log(`Server running on port ${port}`);

console.log(`MongoDB URI: ${mongodbUri}`);

console.log(`Secret Key: ${secretKey}`);

```

By following these steps, you can create and use a `.env` file with environment variables in your Node.js project.

{

"item\_name": "Running Shoes",

"price": 49.99,

"brand": "Nike",

"quantity": 100,

"sizes": [

{ "size": 8, "country": "US" },

{ "size": 9, "country": "US" }

],

"item\_type": "Sports",

"color": "Black",

"material": "Mesh",

"gender": "Men",

"style": "Athletic",

"description": "High-performance running shoes for men."

}

To enhance input sanitization further and integrate express-validator middleware into your application routes, you can follow these steps:

1. Install the necessary packages if you haven't already:

```bash

npm install express express-validator body-parser mongoose bcrypt dotenv

```

2. Update your user routes file (`user\_routes.js`) to include input validation middleware using express-validator:

```javascript

const express = require("express");

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

const bcrypt = require("bcrypt");

const UserModel = require("../models/User"); // Assuming this is your user model

const router = express.Router();

// Validation middleware for creating a new user

const validateNewUser = [

body("name").trim().notEmpty().withMessage("Name is required"),

body("email")

.trim()

.isEmail()

.withMessage("Please provide a valid email address")

.custom(async (value) => {

// Check if email already exists

const existingUser = await UserModel.findOne({ email: value });

if (existingUser) {

throw new Error("Email already exists");

}

return true;

}),

body("password")

.trim()

.isLength({ min: 8 })

.withMessage("Password must be at least 8 characters long")

.matches(

/^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/,

"Password must contain at least one uppercase letter, one lowercase letter, one number, and one special character"

),

body("confirmPassword").custom((value, { req }) => {

if (value !== req.body.password) {

throw new Error("Passwords do not match");

}

return true;

}),

body("userRole").isIn(["Admin", "SalesPerson", "Customer"]),

body("dateOfBirth").isDate().withMessage("Invalid date format"),

body("city")

.trim()

.isLength({ max: 100 })

.withMessage("City cannot exceed 100 characters")

.matches(/^[A-Za-z\s]+$/, "City must contain only alphabetic characters"),

body("country")

.trim()

.isLength({ max: 100 })

.withMessage("Country cannot exceed 100 characters")

.matches(/^[A-Za-z\s]+$/, "Country must contain only alphabetic characters"),

body("phoneNumber")

.trim()

.matches(/^\+\d{1,3}\d{5,15}$/)

.withMessage("Please provide a valid phone number"),

body("gender").isIn(["Male", "Female", "Other"]),

body("profilePicture")

.optional({ checkFalsy: true })

.isURL()

.withMessage("Please provide a valid URL for the profile picture"),

];

// Route to create a new user

router.post("/", validateNewUser, async (req, res) => {

try {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

// Create the new user

const newUser = new UserModel({

name: req.body.name,

email: req.body.email,

password: req.body.password,

userRole: req.body.userRole,

dateOfBirth: req.body.dateOfBirth,

address: req.body.address,

city: req.body.city,

country: req.body.country,

phoneNumber: req.body.phoneNumber,

gender: req.body.gender,

profilePicture: req.body.profilePicture,

});

// Hash the password

const salt = await bcrypt.genSalt(10);

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

// Save the user to the database

await newUser.save();

res.status(201).json({ message: "User created successfully" });

} catch (error) {

console.error(error);

res.status(500).json({ message: "Server error" });

}

});

module.exports = router;

```

In this updated code:

- I've added validation middleware using express-validator for creating a new user.

- The middleware checks and sanitizes user input based on various validation rules.

- Integrated the validation middleware into the POST route for creating a new user.

- Added error handling to return appropriate responses for validation errors and server errors.

Make sure to adjust the validation rules and error handling as per your specific requirements and business logic.

Here's the debugged and rewritten code with adherence to professional standards, security, and technical efficiency:

```javascript

const jwt = require('jsonwebtoken');

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

// Middleware function to protect routes requiring authentication

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

try {

let token;

// Extract token from request headers or cookies

const authHeader = req.headers.authorization;

const cookieToken = req.cookies.token;

// Determine the source of the token (headers or cookies)

if (authHeader && authHeader.startsWith("Bearer")) {

token = authHeader.split(" ")[1];

} else if (cookieToken) {

token = cookieToken;

}

// Ensure token exists

if (!token) {

return res.status(401).json({

status: false,

error: "Unauthorized - No token provided",

});

}

// Verify the token

const decoded = jwt.verify(token, process.env.SECRET\_KEY);

// Find the user associated with the token

const user = await UserModel.findById(decoded.data);

// Ensure user exists

if (!user) {

return res.status(401).json({

status: false,

error: "Unauthorized - User not found",

});

}

// Attach the user to the request object for further use

req.user = user;

next(); // Proceed to the next middleware

} catch (error) {

console.error(error);

res.status(401).json({ error: "Unauthorized - Invalid token" });

}

};

```

Changes made:

1. Added comments to explain the purpose of each section of the code.

2. Renamed variables for clarity and adherence to professional standards.

3. Consolidated error handling to a single catch block for better readability.

4. Removed redundant comments and unnecessary status properties from the response.

5. Ensured proper token handling and verification to enhance security.

6. Simplified logic for extracting the token from headers or cookies.

7. Improved code readability and maintainability by restructuring the code.

To ensure that both the request headers authorization bearer and cookies must return true, you can modify the code as follows:

```javascript

const jwt = require('jsonwebtoken');

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

// Middleware function to protect routes requiring authentication

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

try {

let token;

// Extract token from request headers authorization bearer

const authHeader = req.headers.authorization;

const isBearerToken = authHeader && authHeader.startsWith("Bearer");

// Extract token from cookies

const cookieToken = req.cookies.token;

// Check if both authorization bearer and cookies exist

if (isBearerToken && cookieToken) {

token = authHeader.split(" ")[1];

}

// Ensure token exists

if (!token) {

return res.status(401).json({

status: false,

error: "Unauthorized - No token provided",

});

}

// Verify the token

const decoded = jwt.verify(token, process.env.SECRET\_KEY);

// Find the user associated with the token

const user = await UserModel.findById(decoded.data);

// Ensure user exists

if (!user) {

return res.status(401).json({

status: false,

error: "Unauthorized - User not found",

});

}

// Attach the user to the request object for further use

req.user = user;

next(); // Proceed to the next middleware

} catch (error) {

console.error(error);

res.status(401).json({ error: "Unauthorized - Invalid token" });

}

};

```

Changes made:

1. Separate extraction of token from request headers authorization bearer and cookies.

2. Introduce a boolean variable `isBearerToken` to check if the request contains authorization bearer.

3. Ensure that both authorization bearer and cookies exist before extracting the token.

4. Modified comments to reflect the updated code logic.