The project structure you've outlined follows the **MVC (Model-View-Controller)** design pattern, which is commonly used to separate concerns and promote scalability in web applications. Here's a breakdown of each folder and file in your structure:

**1. config/**

This folder is where configuration files live. The key files would typically handle setting up connections and environment variables.

* **db.js**: This file would typically contain the database connection logic (MySQL or another DB) that will be used across the project.
  + Example of db.js:
  + const mysql = require('mysql2');
  + const connection = mysql.createConnection({
  + host: 'localhost',
  + user: 'root', // Your MySQL username
  + password: '', // Your MySQL password
  + database: 'your\_database\_name'
  + });
  + connection.connect(err => {
  + if (err) {
  + console.error('Error connecting to the database:', err);
  + process.exit(1); // Exit if there's a DB connection error
  + }
  + console.log('Connected to the database.');
  + });
  + module.exports = connection;

**2. controllers/**

This folder contains the **controller** files that handle incoming requests, perform business logic, and return appropriate responses. The controllers interact with models and views to manage data flow.

* **authController.js**: Manages authentication logic (login, registration, session management).
* **budgetsController.js**: Handles requests related to budget management (create, view, update, delete budgets).
* **complianceAuditsController.js**: Manages requests related to compliance audits (view, manage audits).
* **contractsController.js**: Manages the logic for contract creation, tracking, renewal, and management.
* **notificationsController.js**: Handles email notifications, contract expiration alerts, and similar tasks.
* **purchaseOrdersController.js**: Handles operations related to purchase orders.
* **tasksController.js**: Manages the creation and tracking of tasks.
* **vendorsController.js**: Deals with vendor management (register, update, delete vendors).

Each of these files would export functions that handle different HTTP routes (GET, POST, PUT, DELETE).

**3. middleware/**

This folder contains **middleware** files that intercept requests and perform operations before passing control to the route handlers (controllers). For example, you could have middleware for authentication checks or input validation.

* **authMiddleware.js**: Checks whether a user is authenticated or has the correct role to access a specific route.
* **validation.js**: Contains validation logic for incoming requests (e.g., validating if fields like email or contract\_id are correctly formatted).

**4. models/**

This folder is responsible for interacting with the database. Models represent the data structures, and they typically include methods for querying the database.

* **authRoutes.js**: Handles logic related to user authentication routes (could contain helper functions for login, registration).
* **budgetModel.js**: Contains methods for managing budgets in the database (e.g., CRUD operations).
* **complianceAuditModel.js**: Contains methods for compliance audits (e.g., auditing vendors or contracts).
* **contractModel.js**: Manages contract-related data (e.g., creating contracts, retrieving active contracts).
* **notificationModel.js**: Contains logic for managing notifications (email reminders, alert creation).
* **purchaseOrderModel.js**: Interacts with the database to handle purchase order operations.
* **taskModel.js**: Contains logic for managing tasks (e.g., create, update, retrieve tasks).
* **userModel.js**: Represents the user data and handles queries related to users.
* **vendorModel.js**: Contains methods for interacting with the vendor data (e.g., registering new vendors, viewing vendor performance).

**5. node\_modules/**

This folder will be automatically generated when you run npm install. It contains all the project dependencies (e.g., Express, Nodemailer, MySQL2, etc.).

**6. public/**

This folder holds static assets that are publicly accessible (CSS, images, JS files). These files are used by the frontend part of the application.

* **styles.css**: Contains the CSS styles for the frontend user interface.

**7. routes/**

This folder defines the HTTP routes for your application. These files will define the routes and map them to controller functions.

* **auth.js**: Routes for user authentication (login, registration, etc.).
* **budgets.js**: Routes for managing budgets.
* **complianceAudits.js**: Routes for handling compliance audits.
* **contracts.js**: Routes for managing contracts.
* **dashboard.js**: This could be the route for accessing the dashboard page that aggregates data from various modules (like contracts, budgets, tasks, etc.).
* **notifications.js**: Routes related to notification management.
* **purchase-orders.js**: Routes for managing purchase orders.
* **tasks.js**: Routes for managing tasks.
* **vendors.js**: Routes for managing vendors.

**8. views/**

This folder contains **EJS (Embedded JavaScript)** templates, which are used to render dynamic HTML views. EJS allows you to insert JavaScript code into HTML files.

* **404.ejs**: A page to show when a requested resource is not found.
* **500.ejs**: A page for showing server errors.
* **budgets.ejs**: A view for displaying and managing budgets.
* **complianceAudits.ejs**: A view for displaying and managing compliance audits.
* **contracts.ejs**: A view for displaying contracts.
* **dashboard.ejs**: The main dashboard view for your application.
* **index.ejs**: The landing page of the website.
* **login.ejs**: A login page for user authentication.
* **notifications.ejs**: A page to view notifications.
* **purchase-orders.ejs**: A page for viewing and managing purchase orders.
* **register.ejs**: A page for user registration.
* **taskDetail.ejs**: A page for viewing the details of a specific task.
* **tasks.ejs**: A page for managing tasks.

**9. .env**

This file stores environment variables such as database credentials, API keys, and other sensitive information. Example of .env file:

DB\_HOST=localhost

DB\_USER=root

DB\_PASSWORD=

DB\_NAME=your\_database\_name

EMAIL\_USER=your\_email@gmail.com

EMAIL\_PASS=your\_email\_password

**10. app.js**

This is the main entry point of the Node.js application. It sets up Express, middleware, routes, and database connections.

Example of app.js:

const express = require('express');

const app = express();

const mysql = require('mysql2');

const dotenv = require('dotenv');

const path = require('path');

// Load environment variables

dotenv.config();

// Connect to the database

const db = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

database: process.env.DB\_NAME

});

// Middlewares

app.use(express.json());

app.use(express.static(path.join(\_\_dirname, 'public')));

// Routes

const authRoutes = require('./routes/auth');

const budgetRoutes = require('./routes/budgets');

const contractRoutes = require('./routes/contracts');

// Use routes

app.use('/api/auth', authRoutes);

app.use('/api/budgets', budgetRoutes);

app.use('/api/contracts', contractRoutes);

// Error handling (404, 500)

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

res.status(404).render('404');

});

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

res.status(500).render('500');

});

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {

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

});

**11. package.json and package-lock.json**

These files are automatically generated by npm and store information about the project, including dependencies, scripts, and versioning.

**Key Points to Consider**

1. **Separation of Concerns**: With the MVC pattern, your business logic (controllers), data models (models), and views (views) are clearly separated, making the application easier to maintain and scale.
2. **Database Integration**: Using mysql2 in the models/ folder allows you to interact with your database for CRUD operations.
3. **Routing and Middleware**: The **routes/** folder defines all the API endpoints, and **middleware/** handles tasks like user authentication, validation, or logging before reaching the controller functions.

This structure will help keep your code organized and modular, and it's a good foundation for building more features in the future!