**Documentation for MERN Stack Application**

This documentation provides an overview of a MERN (MongoDB, Express.js, React.js, Node.js) stack application, focusing on its structure, functionality, and key features. The application is designed to manage users, logs, and metrics, with a strong emphasis on security, scalability, and performance.

**Table of Contents**

1. **Application Overview**
2. **Backend Structure**
   * Middleware
   * Routes
   * Models
   * Controllers
3. **API Endpoints**
   * User Management
   * Log Management
   * Metrics
4. **Error Handling**
5. **Database Connection**
6. **Key Features**
7. **Usage Examples**

**1. Application Overview**

The application is a backend service built with Node.js and Express.js, using MongoDB as the database. It provides APIs for user authentication, user management, logging, and metrics tracking. The application is secured using Helmet, CORS, and JWT-based authentication. It also includes middleware for role-based access control and activity logging.

**2. Backend Structure**

**Middleware**

The application uses the following custom middleware to enhance functionality and security:

1. **verifyToken**:
   * Verifies the JWT token provided in the Authorization header.
   * If the token is valid, it attaches the decoded user data to the req.user object.
   * If the token is invalid or missing, it returns a 403 or 401 status, respectively.
2. **isAdmin**:
   * Checks if the user has an admin role.
   * If the user is an admin, it allows the request to proceed.
   * If not, it returns a 403 status (Forbidden).
3. **logActivity**:
   * Logs user activities such as API requests.
   * Captures details like the user's IP address, user agent, and request body/query/parameters.
   * Saves the log to the database for auditing and analytics purposes.

**Routes**

The application has the following routes:

* **/api/auth**: Handles user authentication (login, registration).
* **/api/users**: Manages user-related operations (create, get, ban, approve).
* **/api/logs**: Handles log creation and retrieval.
* **/api/metrics**: Provides metrics like active users and page views.

**Models**

The application uses two MongoDB models:

1. **User Model**:
   * Fields: email, password, role, status.
   * Password is hashed before saving.
   * Status can be pending, approved, or banned.
2. **Log Model**:
   * Fields: action, user, createdAt, details, ipAddress, userAgent.
   * Used to track user actions and API requests.

**Controllers**

The controllers handle the business logic for each route:

* **User Controller**:
  + Manages user registration, login, and role-based actions (e.g., approve, ban).
* **Log Controller**:
  + Handles log creation and retrieval.
  + Provides metrics like daily and weekly active users and page views.

**3. API Endpoints**

**User Management**

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Description |
| /api/auth/register | POST | Registers a new user. |
| /api/auth/login | POST | Logs in a user and returns a JWT. |
| /api/users | GET | Fetches users with filters and pagination. |
| /api/users | POST | Creates a new user. |
| /api/users/:id/approve | PUT | Approves a pending user. |
| /api/users/:id/ban | PUT | Bans a user. |

**Log Management**

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Description |
| /api/logs | POST | Creates a new log entry. |
| /api/logs | GET | Fetches logs with filters and pagination. |

**Metrics**

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Description |
| /api/metrics/daily-active-users | GET | Returns daily active users. |
| /api/metrics/weekly-active-users | GET | Returns weekly active users. |
| /api/metrics/daily-page-views | GET | Returns daily page views. |
| /api/metrics/weekly-page-views | GET | Returns weekly page views. |

**4. Error Handling**

The application uses a custom ApiResponse class to handle responses and errors. Errors are logged and returned with appropriate status codes and messages.

**5. Database Connection**

The application connects to MongoDB using Mongoose. The connection string is stored in the environment variable MONGO\_URI.

**6. Key Features**

* **User Authentication**: Secure login and registration using JWT.
* **Role-Based Access Control**: Users can have different roles (e.g., admin, user).
* **Activity Logging**: Tracks user actions and API requests for auditing and analytics.
* **Pagination and Filtering**: Supports pagination and filtering for user and log data.
* **Metrics Tracking**: Tracks daily and weekly active users and page views.
* **Security**: Uses Helmet for HTTP header security and bcrypt for password hashing.

**7. Usage Examples**

**Register a User**

* **Endpoint**: /api/auth/register
* **Method**: POST
* **Request Body**: { "email": "user@example.com", "password": "password123" }

**Login a User**

* **Endpoint**: /api/auth/login
* **Method**: POST
* **Request Body**: { "email": "user@example.com", "password": "password123" }

**Fetch Users**

* **Endpoint**: /api/users
* **Method**: GET
* **Query Parameters**: page, perPage, filters

**Create a Log**

* **Endpoint**: /api/logs
* **Method**: POST
* **Request Body**: { "action": "page\_view", "user": "user\_id" }

**Get Daily Active Users**

* **Endpoint**: /api/metrics/daily-active-users
* **Method**: GET

**Front-End**

**1. Project Overview**

The front-end application is built using **React** as the core framework, with **React-Bootstrap** for UI components and **React Router** for navigation. It incorporates modern development practices such as component-based architecture, modular styling, and efficient state management. The application is designed to be scalable, maintainable, and performant.

**2. Architecture**

The application follows a **modular and component-based architecture**, ensuring separation of concerns and reusability. Key features include:

* **Single-Page Application (SPA)**: Utilizes React Router for seamless navigation.
* **Layered Structure**: Divides the application into layers (e.g., presentation, logic, and data).
* **Responsive Design**: Ensures compatibility across devices using Bootstrap's grid system and utility classes.

**3. Folder Structure**

The project is organized into a well-defined folder structure to enhance readability and maintainability:

* **assets/**: Stores static resources like images, icons, and fonts.
* **components/**: Contains reusable UI components (e.g., buttons, modals).
* **layouts/**: Houses layout components (e.g., header, sidebar, footer).
* **pages/**: Includes page-level components (e.g., Home, Login, Dashboard).
* **styles/**: global and component-specific styles.

**4. Component Design**

Components are designed to be **reusable, modular, and self-contained**. Each component follows these principles:

* **Props-Based Customization**: Components accept props for dynamic behavior and styling.
* **Separation of Concerns**: Logic and presentation are kept separate for clarity.
* **Documentation**: Each component includes a description of its purpose, props, and usage.

**5. Styling Approach**

The application uses a **combination of global and modular styles**:

* **CSS Modules**: Ensures scoped styles to prevent conflicts.
* **Bootstrap**: Provides a responsive grid system and pre-built UI components.
* **Responsive Design**: Media queries and Bootstrap utilities for device compatibility.

**6. Routing Strategy**

Routing is managed using **React Router**, enabling a seamless SPA experience:

* **Protected Routes**: Certain routes are restricted to authenticated users.
* **Dynamic Routing**: Supports nested routes and parameterized URLs.