Village Management System

Documentation

Contents

1. Introduction

1.1 Purpose of the Report

This report provides a comprehensive overview of the Village Management System (VMS), developed as part of a team project. explaining its front-end, back-end, and database setup, as well as the key features and concepts for each section. This guide is designed to make understanding the project's codebase straightforward.

2.1 Overview of the Village Management System (VMS)

The Village Management System (VMS) is a web application designed to streamline the management of various village activities and services. The project offers features such as managing demographic data, gallery sharing, chat functionality, and administrative tools. It provides an intuitive interface for users to interact and manage tasks efficiently.

2. Folder and File Structure Overview

❖ **Main Files**:

- **index.html**: The entry point of the application, setting up a simple page layout.
- **App.js**: The central file linking all components and routing.
- **server.js & websocketServer.js**: Handle backend communication and real-time updates.

❖ **Directories**:

- **assets**: Contains css files for styling and images for visual elements.
- **components**: Holds React components used in multiple areas, including the dashboard, login, signup, and layout (Header, Footer).
- **data**: Includes sample data for user accounts, gallery, and village demographics.
- **pages**: Contains the core sections of the app like Gallery, Overview, *VillageManagement*, and *ChatSystem*.
- **utils**: Utility functions, like managing WebSocket connections.

3. Front-End Structure Overview

The Village Management System (VMS) front-end is a modular design using **React.js** and **CSS Modules** for maintainable and reusable components.

1.3 Landing Page

❖ **Purpose**: Welcoming interface with login and signup functionalities.
❖ **Features**:

- Header and footer with system information.
- Navigation buttons for login/signup pages.

❖ **Technologies**:

- **React.js**: Modular components like Header and Footer.
- **CSS Modules**: Scoped styling ensures maintainability.

2.3 Login and Signup Pages

❖ **Purpose**: User authentication and account creation.
❖ **Features**:

- **Login**: Validates credentials against localStorage.
- **Signup**: Stores new user data in localStorage, avoiding duplicate usernames.

❖ **Concepts**:

- **React's useState**: Tracks input values.
- **Validation**: Ensures data integrity.

3.3 Dashboard and Sidebar

❖ **Purpose**: Central navigation hub post-login.
❖ **Features**:

- Sidebar with links to "Overview," "Village Management," "Chat," and "Gallery."
- Responsive design with dynamic routing.

❖ **Technologies**:

- **React Router**: Smooth transitions without page reloads.
- **Responsive Design**: Sidebar toggles for mobile usability.

4.3 Overview Section

❖ **Purpose**: Displays a summary of village data using charts and maps.
❖ **Features**:

- **Google Maps Integration**: Dynamically fetches village locations via GraphQL API.
- **Charts**:
  - Pie charts for age distribution and gender ratios.
  - Bar charts for population statistics.
- Real-time updates as data is fetched.

❖ **Technologies Used**:

- **React**: Reusable UI components.
- **Chart.js**: Data visualization.
- **Google Maps API**: Geolocation.

5.3 Village Management Section

❖ **Purpose**: Administer village data with CRUD operations.
❖ **Features**:

- **Admins**:
  - Add new villages via a modal.
  - Update demographic data and village details.
  - View, search, and sort villages.
- **Users**:
  - View village list and perform searches.

❖ **Concepts**:

- **State Management**: useState for UI updates.
- **Dynamic Rendering**: Conditional components for admins.
- **Pagination and Sorting**: Improves usability for large datasets.
- **Database**: uses Mysql and server.js.

6.3 Chat System Section

❖ **Purpose**: Facilitates communication between users and admins.
❖ **Features**:

- **User Perspective**:
  - Users can search for specific admins or select from a list to initiate a chat.
  - All messages are displayed in a single group chat format.
- **Admin Perspective**:
  - Admins view only messages relevant to their group and can respond accordingly.
- **Chat Features**:
  - Real-time messaging via WebSocket.
  - Messages persist in localStorage for future reference.

❖ **Key Concepts**:

- **Role-Based Views**: Admins and users have distinct interfaces.

- **WebSocket Integration**: Real-time communication.
- **Search Functionality**: Enables quick filtering of admins.

How It Works

1. **WebSocket Connection**
   The Chat System establishes a WebSocket connection when the user navigates to the chat section. This connection ensures that any message sent by a user is immediately broadcast to all connected clients.
   - The WebSocket URL is managed in the `/utils/websocket.jsx` file.
   - Incoming messages are received and handled by the `ChatMessages.jsx` component, which updates the interface in real-time.
2. **Message Persistence**
   Messages are stored in a `messages.json` file located on the server. This file acts as a lightweight database to save the chat history.
   - When a message is sent, it is appended to the messages.json file.
   - On loading the chat, the file is parsed, and its contents are displayed to users.
3. **Component Breakdown**
   - **`Chat.jsx`**: The parent component that orchestrates the entire chat system by combining all sub-components.
   - **`ChatHeader.jsx`**: Displays the title and basic information about the chat room.
   - **`ChatMessages.jsx`**: Responsible for rendering the list of messages. It subscribes to updates via the WebSocket connection.
   - **`ChatInput.jsx`**: Contains the input field and send button, allowing users to type and submit messages.
   - **`UserList.jsx`**: Displays a dynamic list of all users currently active in the chat.
4. **Real-Time Updates**
   When a user sends a message:
   - The `ChatInput.jsx` component captures the message and sends it through the WebSocket connection.
   - The server broadcasts the message to all connected clients.
   - The `ChatMessages.jsx` component listens for new messages and appends them to the visible message list.
   - Simultaneously, the message is written to the `messages.json` file for persistence.

7.3 Gallery Section

❖ **Purpose**: Manage and display village-related images.
❖ **Features**:

- Dynamically loads items from localStorage.
- Admin-only functionality for adding images via URL or upload.
- Modal for entering image details.

❖ **Key Concepts**:

- **Data Persistence**: Images stored in localStorage.
- **Conditional Rendering**: Admin privileges control visibility of features.
- **File Handling**: Converts uploaded files for storage.

4. Back-End Structure

The back-end is built using **Express.js**, **GraphQL**, and **MySQL** for a scalable and efficient architecture.

1.4 GraphQL API

❖ **Purpose**: Efficiently fetch data with reduced over-fetching.
❖ **Features**:
  - Queries for fetching villages and detailed data.
  - Mutations for adding, updating, and deleting villages.
❖ **Concepts**:
  - **Schema Design**: Defines queries and mutations.
  - **Apollo Client Integration**: Simplifies front-end requests.

2.4 WebSocket Server

- **Purpose**: Enables real-time updates for the chat system.
- **Features**:
  - Dynamic message management.
  - Broadcasts updates to all connected clients.
- **Concepts**:
  - **Persistent Storage**: Saves messages to a JSON file.
  - **Broadcasting**: Sends updates to multiple WebSocket clients.

3.4 Database

- **Current Setup**: Uses **localStorage** for:
  - User data (usernames, passwords, roles).
  - Chat messages (grouped by admin).
  - Gallery items (image details).
- **Future Plans**: Transition to **MySQL** for robust, scalable data management.

4.4 MySQL Integration

- **Purpose**: Stores villages and demographic data.
- **Features**:
  - Tables for village details, demographic data, and tags.
  - Data retrieval and updates via GraphQL resolvers.
- **Concepts**:
  - **Relational Schema**: Defines structured data relationships.
  - **SQL Queries**: Fetch, insert, update, and delete operations.

Thanks for your time ☺