System Architecture Overview

This document provides a medium-level overview of the fire monitoring application's architecture, detailing the main components and how data flows through the system. For this assessment, fire recognition system is built based on buildings to visualise real-case scenario effectively.

Core Components

The application is built with a clear separation between the backend, which handles data and logic, and the frontend, which handles the user interface.

1. Python Backend (FastAPI)

The backend is the brain of the application. It's responsible for managing and providing data.

- **Server:** Built with **FastAPI**, it runs on a dedicated port (e.g., `http://localhost:3001`). Its main job is to listen for requests from the frontend.
- **Sensor Simulation:** The `simulate_fire_events()` function runs as a background task. This is the source of all the data. It mimics a real-world sensor system by using Python's `random` module to generate a new fire alert every few seconds. Each simulated alert includes a random `location`, `intensity` level (from 1 to 10), an `active` status, and a valid `timestamp`.
- **Alert Storage:** A simple in-memory dictionary (`active_fire_alerts`) stores the currently active alerts. This data is not persistent, meaning it resets when the server is stopped.
- **API Endpoint:** The `get_fire_alerts()` function serves as the **API endpoint**. When the frontend makes a request to this endpoint, the backend responds with the latest list of active alerts in JSON format.

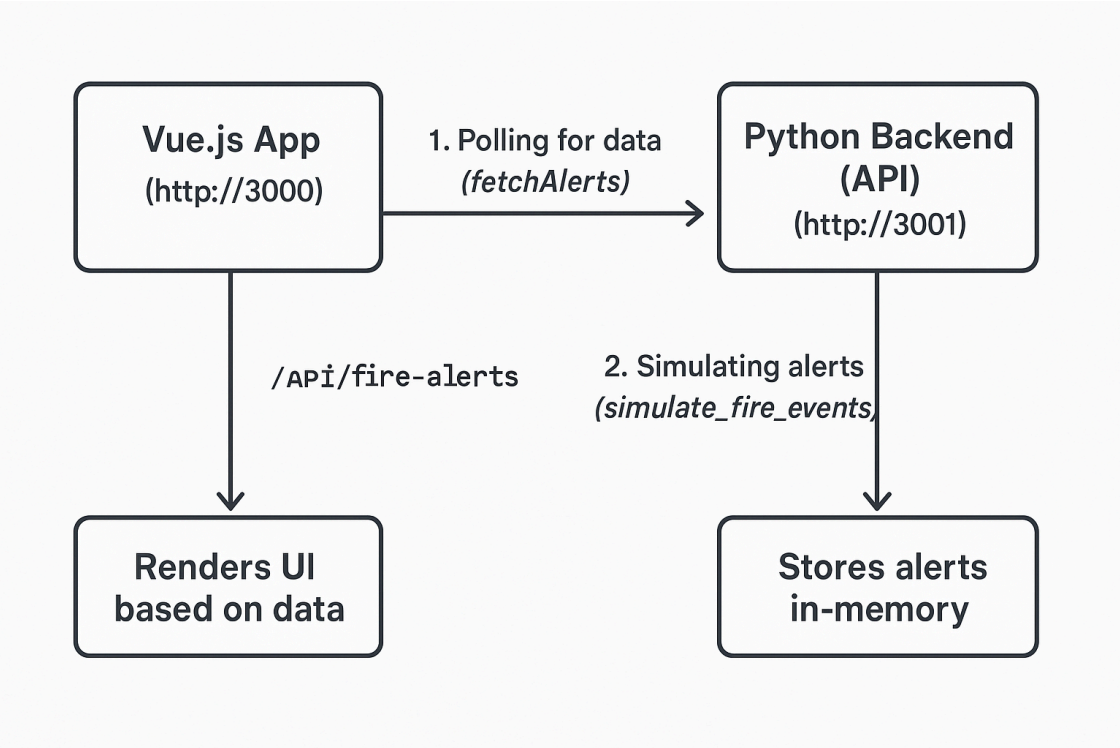
2. Frontend UI (Vue.js)

The frontend is the visual dashboard that users interact with. It consumes the data provided by the backend to display information in an organized way.

- **Dashboard:** Built with **Vue.js**, it runs on a separate port (e.g., `http://localhost:3000`). It is responsible for rendering the grid of locations and displaying alert details.
- **Data Fetching:** The `fetchAlerts()` function is the key to connecting the frontend to the backend. It uses a timer to **periodically poll** the backend's API endpoint every few seconds to get the most up-to-date information.
- **Reactive State:** The application uses Vue's **reactive state** (e.g., `fireAlerts`) to automatically update the user interface whenever new data is fetched from the backend.
- **Visual Intensity Indicators:** To provide a quick and intuitive understanding of an ongoing incident's severity, the dashboard utilizes a color-coding system based on the `intensity` value from the simulated alerts. This helps operators immediately prioritize and assess threats.
    - **Low Intensity (1-4):** Represented by a **yellow** highlight, indicating a minor incident that requires attention.
    - **Medium Intensity (5-7):** Displayed with an **orange** highlight, signaling an escalating situation that needs immediate action.
    - **High Intensity (8-10):** Shown with a pulsing **red** highlight, signifying a critical threat that demands urgent intervention.
- **User Interface Logic:** The frontend also contains the logic for handling user interactions, such as clicking on a zone to view alert details and showing the simulated "Send Alert" modal.

Architectural Flowchart

This flowchart illustrates the primary flow of data and communication between the frontend and backend components.



Flow Description

1. A background task in the **Python Backend** continuously generates new, random fire alerts.
2. The backend stores these active alerts in a simple in-memory dictionary.
3. The **Vue.js Frontend**, when loaded, initiates a timer that repeatedly calls the `get_fire_alerts` API endpoint on the backend.
4. The backend receives the request and returns a JSON list of all the active fire alerts it has in its memory store.
5. The frontend receives this JSON data and updates its reactive state, which in turn causes the user interface to automatically render the changes, highlighting active fire zones in the dashboard.