**Architectural Description**

# **SafeZones Product Vision:**

Empower individuals and communities to stay informed and safe by enabling real-time reporting and sharing of potentially hazardous events through an intuitive, map-based mobile application. This tool is designed for local residents and tourists alike, providing them with a reliable platform to identify, share, and discuss safety-related incidents. By using location-based alerts, categorized event markers, and a community-driven feedback system, users can proactively navigate urban environments with greater awareness and security.

**Synthesis:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Functionality** | | |  | | --- | | **Intermediate Deliverable** | | |  | | --- | | **Final Product** | |
| Interactive Incident Mapping | Users can mark incidents on a map, categorizing them as medium (yellow) or severe (red), and place them within a 300m radius with a short description. | **Complete.** Users can still report incidents with the same categorization and location radius, ensuring real-time situational awareness. |
| Diverse Incident Categories | Users can select from multiple categories such as Assault, Harassment, and Robbery to ensure accurate classification. | **Complete.** All categories remain available, allowing users to classify incidents precisely. |
| Timed Pin Visibility | Medium-severity incidents remain visible for 12 hours, while severe incidents last for 24 hours before being removed. | **Complete.** The same time-based visibility system is retained, ensuring timely and relevant incident reporting. |
| User-Driven Engagement | Users can receive likes on their reports, helping them build credibility and level up within the app. | **Complete.** The engagement system remains unchanged, encouraging active participation and trust-building. |
| Detailed Incident Info | Each incident includes information such as category, time posted, user trust factor, description, and number of likes. | **Complete.** Users still have access to comprehensive incident details, allowing them to assess risk effectively. |
| Dynamic Incident Navigation Bar | Users can view a list of incidents within 0.5 km for medium-severity cases and 1 km for severe ones. | **Complete.** The navigation bar remains functional, helping users stay informed about nearby incidents. |
| Incident Messaging | Users can send and receive messages related to specific incidents directly within the app. | **Complete.** The messaging system remains fully available, allowing seamless communication between users. |
| Contact Support | Users can reach out for assistance through the app’s support feature. | **Complete.** The support function remains available, ensuring quick access to help when needed. |
| Notification System | Users receive push notifications to stay informed about incidents and updates. | **Modified.** Instead of push notifications, alerts are now only visible within the app, requiring users to open it to see updates. This change may impact real-time responsiveness, as users will not receive external alerts. |
| Trusted Contacts | Users can add trusted contacts to their network for added safety and community support. | **Complete.** The trusted contacts feature remains unchanged, enabling users to connect with reliable individuals in case of emergencies. |

**Non-functional Requirements and Architectural Solutions**

1. Performance:

* Non-Blocking UI: Network calls (\_fetchMarkers, \_fetchTrustedUsersLocations) and location updates are handled asynchronously, preventing UI freezes during data loading.
* Parallel Icon Loading: Future.wait in \_fetchMarkers efficiently loads multiple marker icons in parallel rather than sequentially.
* No Session Overhead: The controller is stateless, relying on Spring’s singleton scope, which reduces memory overhead and improves scalability under high concurrency.
* Selective Data Exposure: Methods like getLocations return only essential fields (username, latitude, longitude) instead of full entities, reducing network payload size.

1. Scalability:

* Modular & Well-Defined Endpoints
* The API follows RESTful principles, making it easy to extend and maintain. Each functionality (e.g., user management, contacts, location updates) is clearly separated.
* Efficient Database Access Uses @Transactional annotation for batch updates, reducing multiple database calls and improving performance. The Optional<User> approach prevents unnecessary lookups and reduces potential errors.

1. Security

* Google Authentication: This is a secure and convenient method since users don’t need to manage separate passwords. Google handles authentication, reducing risks like password reuse and phishing.
* Email Verification: This prevents unauthorized access using fake or temporary emails, ensuring users own the accounts they register.
* Password Hashing: If you’re hashing passwords before storing them (Firebase's built-in hashing), this protects against credential leaks, making it difficult for attackers to recover plaintext passwords.

1. Usability

* User-friendly UI/UX design with clear navigation and intuitive interaction.
* Dark mode and high-contrast UI for better accessibility.
* **Responsive design** to ensure the application adapts well to various screen sizes and devices.

1. Extensibility

* Design UI components to be fully modular and reusable, ensuring they can be swapped or updated independently.
* The frontend and backend (Node.js + Firebase) are deployed separately, meaning they function independently but communicate through API requests.

1. Errors

* Implement **centralized logging** with **Firebase Crashlytics** or **Sentry** to track real-time errors.
* Implement proper handling for HTTP response codes (e.g. 404, 500).
* Store user actions in a **queue** and retry them later.

### **QA:**

### Testing for User Management

* **Testing Objective:** Verify the process of user creation, retrieval, and data integrity.
* **Testing Phase:** **Development** – ensuring that core user operations function correctly before integration.
* **Testing Method:** **Unit tests using JUnit and Mockito**, simulating database interactions and validating expected responses.
* **Results:** User-related functionalities operate as expected, correctly handling retrieval and account creation. However, update and deletion processes lack test coverage and should be included in future iterations.

1. Testing for Data Submission and Retrieval

* **Testing Objective:** Ensure that user-submitted data is processed, stored, and retrieved correctly.
* **Testing Phase:** **Development** – validating database interactions before integrating with other components.
* **Testing Method:** **Unit tests with mock repositories**, verifying that the system correctly handles saving and retrieving structured data.
* **Results:** The system successfully processes and returns stored information. However, additional validation is needed for incomplete or incorrect data inputs.

1. Testing for User Authentication and Navigation

* **Testing Objective:** Evaluate the reliability of the authentication process and the fluidity of user navigation throughout the application.
* **Testing Phase:** **Final development stage** – ensuring a smooth user experience before broader testing and deployment.
* **Testing Method:** **Hands-on testing by developers**, logging in through both Google Sign-In and email/password authentication while assessing UI behavior across different devices.
* **Results:** Authentication works as expected, granting appropriate access to users. However, minor layout inconsistencies were identified on smaller screens, necessitating UI refinements for better responsiveness.

### Testing for Multi-Platform Compatibility

* **Testing Objective:** Ensure a consistent user experience across different devices, screen sizes, and operating systems.
* **Testing Phase:** UI/UX validation in the final stages before launch.
* **Testing Method:**Manual testing on multiple devices (Android, iOS).
* **Results:** The app runs well on all devices.

1. Testing for Incident Reporting and Pin Management

* **Testing Objective:** Verify that users can accurately place and manage incident pins.
* **Testing Phase:** Feature validation before release.
* **Testing Method:**Manual testing by placing pins and verifying color categorization.
* **Results:** Pin placement works as expected.

#### **Testing for Incident Messaging System**

* **Testing Objective:** Ensure real-time chat functionality within incident reports.
* **Testing Phase:** Feature validation before deployment.
* **Testing Method:**Manual testing by sending and receiving messages across different users.
* **Results:** Messages are transmitted successfully in real-time, with all user interactions properly synchronized.

#### **Testing for Notification System**

* **Testing Objective:** Ensure timely and relevant in-app notifications.
* **Testing Phase:** Pre-release testing.
* **Testing Method:**Real-device testing to confirm correct display and responsiveness.
* **Results:** Notifications are delivered accurately and consistently, ensuring users receive important updates within the app.

### **Testing for Trusted Contacts System**

* **Testing Objective:** Ensure that users who have been added as trusted contacts can accurately view the location of the person who added them.
* **Testing Phase:** Feature validation before deployment.
* **Testing Method:**
  + **Manual testing** by adding a user as a trusted contact and verifying that they can see the other user's location.
  + **Scenario testing** to ensure location sharing is only visible to authorized contacts.
* **Results:** The trusted contacts feature functions as expected, allowing added users to view location data with proper access control and visibility restrictions.

**Security Analysis:**

* We secure our database by using Spring Data JPA, parameterized queries, and strict input validation. By avoiding raw SQL execution and enforcing best practices, we ensure safe and reliable data handling.
* To enhance security, API keys are stored in a separate environment file instead of being hardcoded in the source code. This prevents unauthorized access, reduces exposure risks, and ensures safer application deployment.
* Our application supports **two authentication methods** managed through **Firebase Authentication**: users can log in **via Google Sign-In** for a seamless experience or create a **separate account with email and password**. Firebase ensures secure user management, session handling, and data protection.

**CI/CD:**

We utilized dedicated **development environments**, allowing each developer to test the latest additions within a **local network** before they progress further. The **staging environment** serves as an intermediate step between development and production, where every team member thoroughly evaluates the newest updates. Here, both technical issues and user experience concerns can be identified and addressed.

Finally, the **production environment** is the ultimate stable version, containing only thoroughly tested and verified changes from staging. Updates are merged into production only once they meet all quality and performance standards, ensuring a reliable and seamless user experience.

### **Environment Configurations**

#### **API Endpoints:**

* **Development:** DEV\_API\_URL: http://localhost:8080/
* **Staging:** STAGING\_API\_URL: https://staging.api.example.com/
* **Production:** PRODUCTION\_API\_URL: https://api.example.com/

### **Development Environment:**

* Debugging enabled for identifying and resolving errors.
* Detailed logs for tracking code execution.
* Use of mock data or local databases for testing.
* Relaxed performance configurations (e.g., higher timeouts).
* Connections to external services (APIs, databases) using test credentials.
* **Database:** Local databases (MySQL) with frequent migrations and resets.

### **Staging Environment:**

* Debugging disabled or limited.
* Logs are less detailed but sufficient for monitoring.
* Uses data and services that closely resemble production.
* Performance configurations similar to production.
* Connections to external services using staging credentials.
* **Database:** Dedicated staging databases with representative test data and automated migrations (but no frequent resets).

### **Production Environment:**

* Debugging completely disabled.
* Minimal logs focused on critical errors and key events.
* Uses real, live data and services.
* Optimized performance configurations (e.g., caching, load balancing).
* Connections to external services using production credentials.
* **Database:** Production databases with replication and regular backups.
* Data is real and sensitive, requiring strict security measures.
* Migrations are manual or semi-automated, with thorough testing before deployment.