# Software Design Document

Quake Alert – An Earthquake Alert System

**Team Members:**

- İsmail Altar Büyükdoğan

- Talha Çetin

- Samet Bozdoğan

- Yakup Çakır

**Document-Specific Task Matrix:**

See ‘docs/TASK_MATRIX.md’.

# Table of Contents


- System Overview
- System Context
- Key Features and Functionality
- Assumptions and Dependencies
- Architectural Design
- Component Design
- Data Design
- Design Patterns
- Implementation Notes
- User Interface Design
- External Interfaces
- Performance Considerations
- Error Handling and Logging
- Design for Testability
- Deployment and Installation Design
- Change Log
- Future Work / Open Issues

# System Overview

Quake Alert is a mobile-first earthquake alert and safety information application built with React Native using the MVVM architectural pattern. It aims to provide real-time (simulated) earthquake alerts to users, display historical earthquake data, and offer interactive earthquake safety tips.

In addition to the mobile frontend, the project includes a backend simulation written in TypeScript, where the Observer Design Pattern is demonstrated in a console-based earthquake detection system. This backend component simulates a central EarthquakeSensor subject that notifies multiple observers such as a NotificationService, LoggerService, and MapUpdaterService. The mobile application and backend simulation are separated but conceptually aligned, allowing for a future version of the app that integrates real earthquake data.

The project is modular, extensible, and prepared for academic presentation, educational purposes, and future production adaptation.

# System Context

Quake Alert operates in two primary contexts:

**Mobile Frontend (React Native App):** Provides a clean, interactive UI to end users. It simulates receiving earthquake data, processes it through ViewModels (using React hooks), and displays the data in three main screens — live alerts, history, and safety tips. All state and logic are separated through MVVM design, making the frontend highly testable and modular.

**Backend Simulation (Node.js + TypeScript):** Designed for demonstrating the Observer Pattern. The backend has no network connection, database, or real sensors — all earthquakes are simulated manually. The architecture emphasizes decoupling through subject/observer interfaces.

The project does not currently integrate real-time APIs or external services but is structured in a way that such integration would be straightforward.

# Key Features and Functionality

**Backend Simulation Features**

- Simulates earthquake events with attributes such as location, magnitude, depth, and timestamp
- Uses the Observer Pattern to notify multiple services:
    - NotificationService: Simulates sending alert messages to users
    - LoggerService: Prints earthquake logs to the console
    - MapUpdaterService: Pretends to update a map visualization layer
- Observers can be dynamically added or removed
- Earthquake data is passed via an update() method call to all registered observers

**Frontend Mobile App Features**

- Real-time simulation of earthquake alerts using a mocked EarthquakeService
- Historical alert listing, with pull-to-refresh and detailed modal views
- Interactive safety tips with categorized instructions for before, during, and after an earthquake
- Multilingual support (English and Turkish) powered by i18next
- Local data persistence using AsyncStorage
- Functional UI with React hooks and modular components
- Color-coded severity levels for earthquake magnitude
- Command Pattern for handling UI interactions in the safety tips modüle

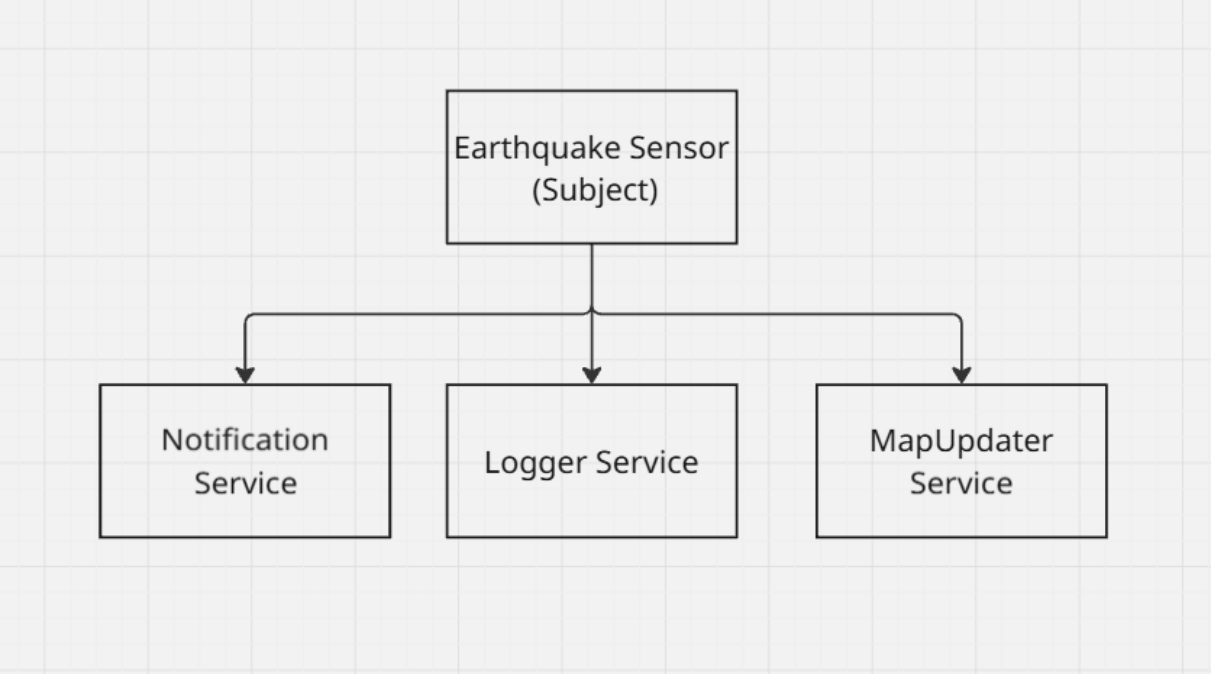
# Assumptions and Dependencies

- The earthquake data is not coming from a real API. It is created manually in the simulation.
- All backend components are trusted and assumed to not fail during execution.
- There is no database, and no external services involved.
- Backend simulation does not include exception handling or persistence.
- The frontend is based on Expo CLI and uses dependencies such as:
    - react-navigation
    - @react-native-async-storage/async-storage
    - i18next and react-i18next
    - react-native-vector-icons, expo-linear-gradient, etc.
    - - expo-notifications
    - - expo-device
    - - react-native-safe-area-context
    - - @expo/vector-icons
- Node.js and ts-node are required for running the backend simulation.

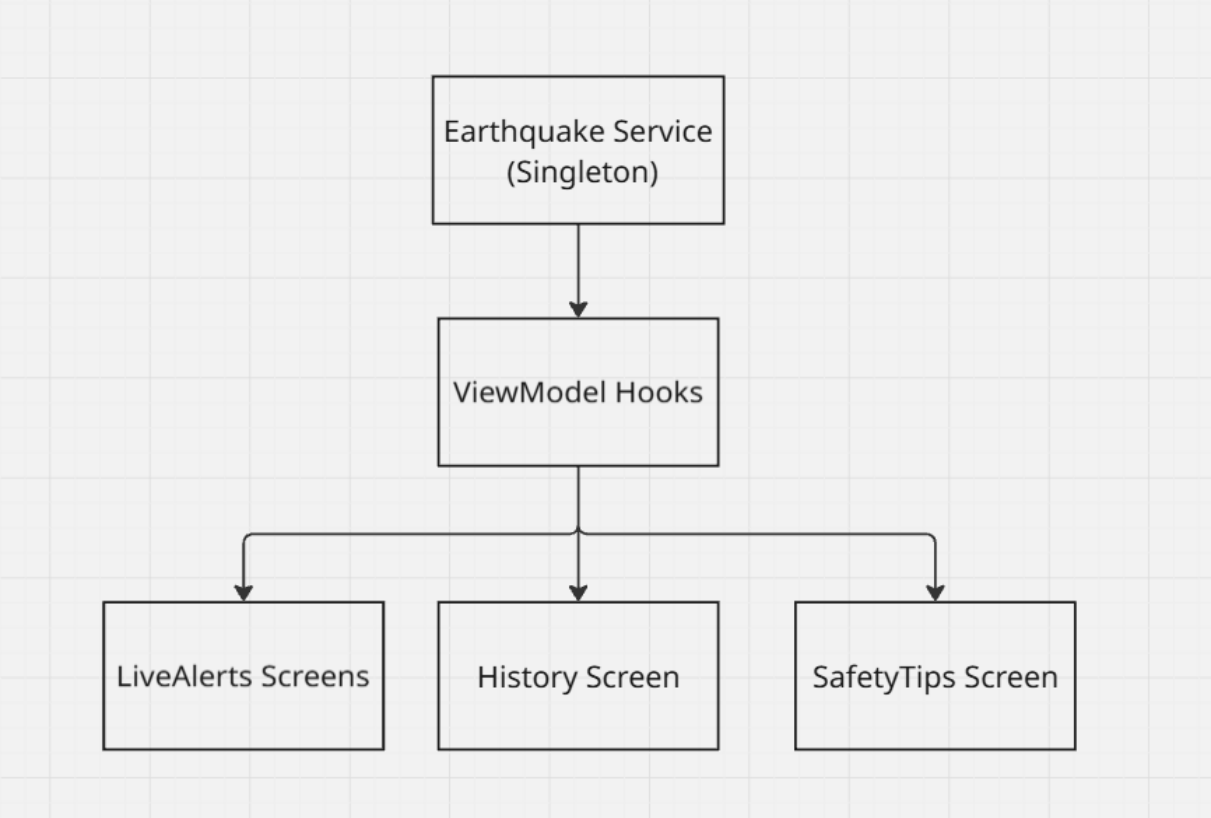
# Architectural Design

## System Architecture (High-Level)

### Backend:




### Frontend:



## Architectural Patterns and Styles

- Observer Pattern (backend): For broadcasting events to multiple observers
- MVVM Pattern (frontend): ViewModel handles logic, View displays UI
- Command Pattern: Used in safety tips module to encapsulate tip interactions
- Singleton Pattern: Shared service instances (e.g., EarthquakeService, StorageService)

## Rationale for Architectural Decisions

The Observer Pattern was chosen for the backend to demonstrate how components can be decoupled yet still respond to shared events. MVVM in the frontend encourages a clean separation of concerns, makes the UI reactive and testable, and simplifies state management.

Command and Singleton patterns increase the maintainability and modularity of shared service logic and user actions.

# Component Design

## Subsytems and Modules

### Backend:

- EarthquakeSensor (Subject): Detects earthquakes and notifies all registered observers.
- NotificationService (Observer): Prints a message that simulates alerting the users.
- LoggerService (Observer): Logs earthquake data with details like location and severity.
- MapUpdaterService (Observer): Pretends to update a map by storing the event as a marker.

### Frontend:

- models/
    - EarthquakeAlert: Represents an alert's magnitude, location, timestamp
    - SafetyTip: Contains step-by-step instructions for different situations
- services/
    - EarthquakeService: Checks every 45 seconds with 30% probability
    - StorageService: A wrapper around AsyncStorage
- viewmodels/
    - useLiveAlertsViewModel, useHistoryViewModel, useTipsViewModel: Provide state and logic to screens
- views/
    - LiveAlertsScreen, HistoryScreen, SafetyTipsScreen: Presentational UI components
- i18n/
    - Language support configuration and translation files (en.json, tr.json)

## Responsibilities of Each Component

### Backend Components:

- EarthquakeSensor: Keeps a list of observers and calls their ‘update()’ methods when an earthquake is detected.
- Observers: Each one focuses on one task and handles the update in its own way

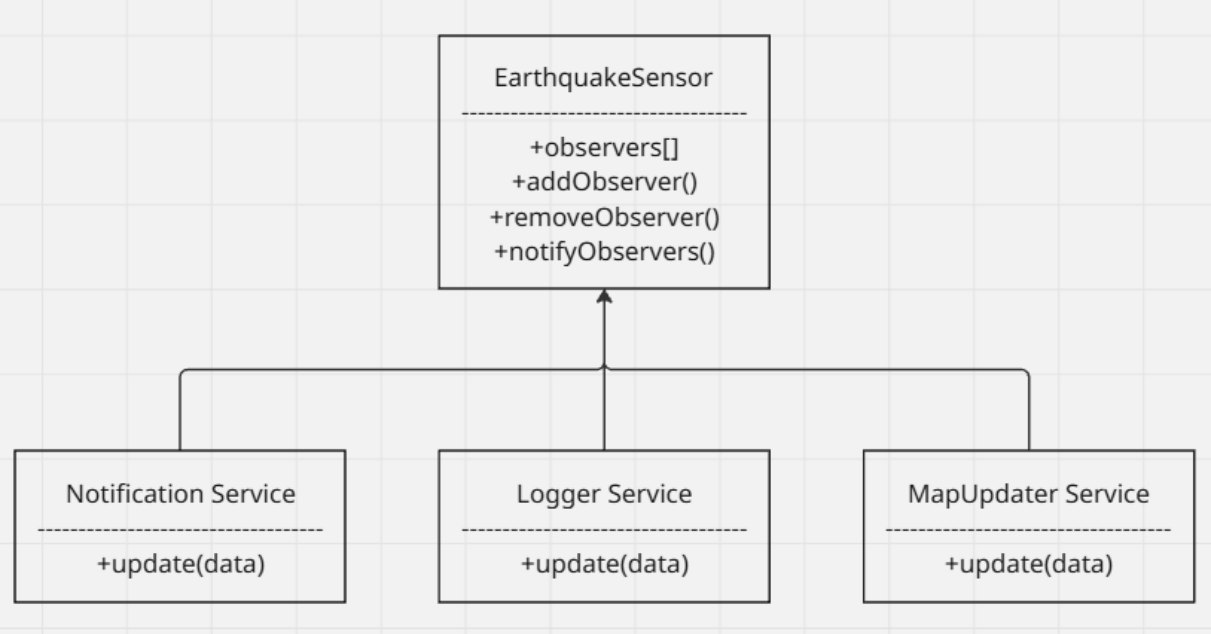
### Frontend Components:

- EarthquakeService: Generates earthquake events (mocked) and makes them accessible to ViewModels.
- StorageService: Interfaces with AsyncStorage to save/retrieve earthquake history and user preferences.
- ViewModels (Hooks): Encapsulate logic for each screen, including state management, formatting, and actions.
- Views (Screens): Display data and provide user interface elements for interaction.
- Models: Represent structured data entities shared across layers.
- i18n system: Handles language selection and dynamic text rendering.

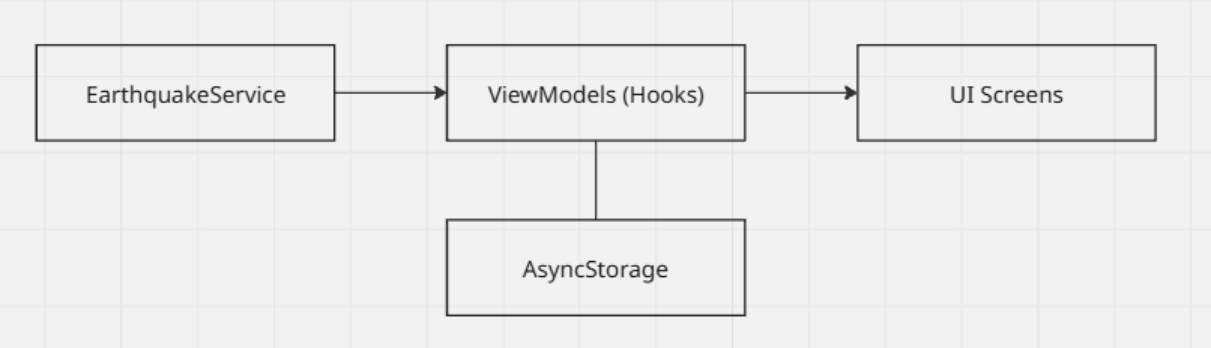
## Interfaces Between Components

- Observers implement IObserver.update(data).
- The subject implements ISubject.addObserver(), removeObserver(), notifyObservers()..
- ViewModels call service methods and expose derived state to views.

## Component Diagrams




## Frontend Component Diagram (simplified)



# Data Design

## Data Models

```
interface EarthquakeData {
 magnitude: number;
 location: string;
 timestamp: Date;
 depth: number;
}

class EarthquakeAlert {
 constructor(id, magnitude, location, timestamp, description = '')
}
class SafetyTip {
 constructor(id, title, description, steps = [])
}
```

## Data Storage

- Backend: In-memory storage only, printed to console
- Frontend: Local storage using AsyncStorage to persist alerts and user settings

# Design Patterns

## Applied Design Patterns

| Pattern | Concrete Use | Key Elements | Responsibility |
|---|---|---|---|
| Observer | EarthquakeService | subscribe(), unsubscribe(), notifyObservers() | Broadcast live-quake data to any listening view-models or UI components without tight coupling |
| Command | useTipsViewModel | createShowTipCommand(), createHideTipCommand(), executeTipCommand() | Encapsulate modal actions so they can be queued, logged, or undone/redone later |
| Singleton | earthquakeService, notificationService | Single exported instance | Guarantee one global point of truth for alert data & push-notification scheduling |
| MVVM | Entire screen stack | Model EarthquakeAlert, SafetyTip View components in views/ ViewModel custom hooks in viewmodels/ | Separate UI, state, and business logic; enable unit-testing of view-models |
| Factory | generateRandomAlert() | Builds EarthquakeAlert objects on demand | Centralise creation logic for mock/test data, easing future changes |
| Strategy | Magnitude colour/icon helpers | getMagnitudeColor(), getMagnitudeIcon() | Swap-in algorithms for visual feedback based on earthquake severity |

## Context and Justification

| Pattern | Problem Addressed | Why This Pattern Fits | Benefits in This App |
|---|---|---|---|
| Observer | Multiple screens (Live list, History, push banner) must react to the same live-alert feed. | Observer decouples EarthquakeService (data producer) from consumers. | - Loose coupling<br>- Easy to extend with new listeners<br>- Simplifies real-time updates |
| Command | Modal hide/show logic quickly became complex and hard to track. | Encapsulating each action as a command keeps state changes predictable and enables a future undo stack. | - Reusable actions<br>- Unit-testable commands<br>- Cleaner view-model code |
| Singleton | Creating new EarthquakeService or NotificationService instances would fragment state and waste memory. | Exporting a single instance provides a controlled global access point. | - Consistent state<br>- Memory efficiency<br>- Simpler dependency injection |
| MVVM | Merging UI and business logic hindered test coverage and readability. | MVVM divides concerns: View = UI, ViewModel = state, Model = data. | - High testability<br>- Clear architecture<br>- Easier onboarding for new devs |
| Factory | Test data needed many EarthquakeAlert variants; manual creation was repetitive. | A factory centralises complex creation and hides construction details. | - DRY code<br>- Simple mock-data generation<br>- One point to change if model evolves |
| Strategy | Hard-coded if/else for magnitude colours/icons violated the open/closed principle. | Strategy lets us plug in new visuals without touching existing logic. | - Cleaner rendering code<br>- Easy to extend (e.g., add shape or sound strategies)<br>- Better UX consistency |

# Implementation Notes

The mobile app is built with React Native 0.79.5 on Expo SDK 53 and follows the MVVM pattern (functional views + hooks-based view-models). Core points:

| **Area** | **Details** |
|---|---|
| Language | JavaScript (ES2022) |
| State | Local component state via useState, lifecycle side-effects via useEffect |
| Navigation | @react-navigation v6 BottomTabNavigator (Live > History > Tips) |
| Design Patterns | Observer (EarthquakeService publishes mock quakes)<br>Command (encapsulated actions in SafetyTipsScreen) |
| i18n | i18next with runtime locale detection |
| Component Style | Pure functional components, colocated styles, test-friendly props |
| Folder Layout | models/, services/, viewmodels/, views/, i18n/ - mirrors the Component Design chapter |

# User Interface Design

## Visual Language & Accessibility

- Primary accent #FF6B35
- Magnitude colouring (≤3 green, 3–5 orange, 5–6 red-orange, 6–7 red, >7 purple); aligns with the colour-coding rationale noted earlier.
- System fonts with dynamic type; all interactive elements meet WCAG 2.1 AA contrast.
- SafeAreaProvider ensures notch / status-bar friendliness on iOS & Android.

## Layout

- Live – real-time feed (FlatList) + severity badge
- History – infinite scroll past events, modal detail
- Tips – card stack with Command-triggered step reveal

# External Interfaces

## APIs

**Expo Notifications:** manages push-alert scheduling and delivery.

**Expo Device:** exposes device model, OS, and locale for analytics and conditional logic.

**AsyncStorage :** provides lightweight on-device key-value persistence.

## Libraries

**React Navigation:** drives screen and tab navigation.

**i18next:** supplies runtime localisation (TR / EN).

## Services

**Expo Push-Notification service:** cloud relay used to reach devices when the app is closed or in the background.

All external calls are wrapped by either EarthquakeService (alerts) or StorageService (local data). During tests these wrappers are mocked, keeping unit tests deterministic.

# Performance Considerations

## Performance Requirements

- Cold start <= 3 seconds.
- Screen change <= 500 milliseconds.
- Push delivery latency <= 1 second.
- Memory kept in check through FlatList-style virtualisation.

## Scalability & Optimization Strategies

- Virtualised Live and History lists to render only what is visible.
- Lazy-load non-critical screens and assets.
- Use React.memo and useCallback to suppress needless re-renders.
- Rely on Expo’s automatic bundle splitting for smaller OTA updates.

# Error Handling & Logging

**Service layer:** try / catch guards around network and storage calls; falls back to cached or mock data if offline.

**UI layer:** a global React Error Boundary swaps in a blank-state component when a render error occurs.

**User feedback:** critical failures trigger a descriptive alert dialog.

**Logging:** console output in development; Expo crash reporting plus a lightweight custom logger in production.

# Design for Testability

Clear View > ViewModel > Service separation eliminates hidden dependencies.

Services are injected, so tests can swap in mocks or stubs.

EarthquakeService.getMockAlerts() provides repeatable sample data.

Helpers are pure functions, enabling straightforward unit tests.

Screens and components mount cleanly with Jest + React Native Testing Library for isolation testing.

# Deployment and Installation Design

**Development**

Runs locally with Expo CLI and hot-reload (.env.development).

Debug builds use the Expo Go client on real devices and emulators.

**Staging / Internal QA**

Builds are generated with eas build -p android and eas build -p ios.

Artifacts are shared with testers through Expo Go links; no public store listing yet.

**Production**

The app has not been submitted to Google Play or the App Store.

# Change Log

v1.0.0 – 23 Jul 2025

- Initial release with Live, History, and Tips screens.
- Mock push-alert pipeline.
- TR / EN localisation.
- Magnitude-based colour scale for alerts.

# Future Work / Open Issues

**Data:** swap mock alerts with a live earthquake API and add GPS-based proximity filtering.

**User preferences:** implement per-user alert thresholds along with quiet-hours support.

**Maps:** integrate offline tile caching and marker clustering.

**Performance:** add background synchronisation plus an Android foreground service.

**Security:** encrypt stored data and optionally support biometric unlock.

**Analytics:** gather anonymised usage metrics to guide future improvements.

**CI/CD:** establish a GitHub Actions → EAS Build pipeline feeding TestFlight and internal testing tracks.

**Known gaps:** the current build still lacks robust offline support, thorough edge-case error handling, and real-time synchronisation.