

DISEÑO E IMPLEMENTACION DE SERVICIOS BACKEND PARA SISTEMA DE COMUNICACION EFECTIVA ANTE CATASTROFES – ALERTIFY

Cristian González Johnson

cristian.gonzalezjo@sansano.usm.cl

Carlos Altén
Profesor Guía

Resumen: Este trabajo presenta el diseño e implementación de un sistema backend para alertas de emergencia que aborda las limitaciones del Sistema de Alerta de Emergencia (SAE) chileno. La solución propuesta utiliza una arquitectura modular basada en Django que integra geolocalización, notificaciones push y almacenamiento híbrido (SQLite/Firebase) para proporcionar alertas contextualizadas y georreferenciadas. El sistema implementa patrones de tolerancia a fallos como Circuit Breaker y ofrece APIs REST para organismos autorizados, optimizando la respuesta ante catástrofes naturales.

Palabras Clave: Sistema de alertas, emergencias, backend, geolocalización, Firebase, Django, tolerancia a fallos

1 Introducción

1.1 Contexto y antecedentes

Los desastres naturales, como incendios forestales, terremotos y tsunamis, representan una creciente amenaza global. En las últimas décadas, factores como el cambio climático y la expansión urbana descontrolada han incrementado significativamente el impacto de estos eventos. En Chile, país particularmente vulnerable, los incendios forestales han devastado vastas áreas, mientras que las constantes amenazas sísmicas y de tsunamis dificultan la planificación y gestión de emergencias.

Según Conaf en la temporada 2023-2024 fueron registrados 5846 incendios. Entre ellos sobresale el mega incendio del Complejo Las Tablas - Reserva Nacional Lago Peñuelas que afectó a 9251 hectáreas en zonas de interfaz urbano forestal, cerca de 15500 viviendas afectadas y 131 fallecidos, la mayor emergencia natural desde el terremoto del 2010. [1]

El Sistema de Alerta de Emergencia (SAE), implementado oficialmente en Chile en el año 2017, representa el esfuerzo institucional más significativo en esta materia. Este protocolo, integrado en todos los teléfonos móviles comercializados en el país, permite a las autoridades enviar notificaciones masivas mediante mensajes de texto y una alerta sonora distintiva. Su principal ventaja radica en la capacidad de operar a través de una red celular, lo que le permite llegar a los dispositivos incluso en casos de una señal de datos deficiente. El SAE se ha consolidado, así como un componente crítico dentro del ecosistema nacional de protección civil. [2]

1.2 Definición del problema

A pesar de la implementación del Sistema de Alerta de Emergencia (SAE) y los avances en gestión de riesgos, persisten limitaciones críticas que comprometen la efectividad de la respuesta ante



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: DISEÑO E IMPLEMENTACIÓN DE SERVICIOS BACKEND PARA SISTEMA DE COMUNICACIÓN EFECTIVA ANTE CATÁSTROFES – ALERTIFY

Nombre del candidato(a): Cristian Andrés González Johnson

Carrera / Grado: Ingeniería en Informática

Campus: Viña del mar **Departamento:** Electrotecnia e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Carlos Alten López, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente

DEJO CONSTANCIA que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 16/01/20226 **Firma:**

Estudiante o Candidato(a):

Fecha: 30/12/2025

Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

emergencias. Los principales problemas que enfrenta la población durante las catástrofes naturales incluyen:

Problemas de comunicación y alertas:

- **Alertas masivas indiscriminadas:** Las notificaciones del SAE se envían a toda la población en un radio amplio sin discriminar el nivel de riesgo real, generando pánico generalizado y evacuaciones innecesarias, especialmente problemático para adultos mayores y personas con condiciones médicas.
- **Limitación de información por restricción de caracteres:** El límite de 90 caracteres del SAE obliga a mensajes extremadamente sintéticos que omiten detalles críticos como la ubicación exacta del peligro, su magnitud, dirección de avance y acciones específicas recomendadas.
- **Falta de contextualización temporal y espacial:** Las alertas no proporcionan información sobre la proximidad real del peligro, tiempo estimado de llegada del evento o duración esperada de la emergencia.

Problemas de evacuación y movilización:

- **Ausencia de guías de evacuación específicas:** Las alertas no indican hacia dónde evacuar, qué rutas utilizar o dónde ubicar centros de refugio seguros, dejando a la población sin orientación práctica para actuar.
- **Congestión vehicular descontrolada:** La evacuación simultánea y descoordinada genera colapsos de tráfico que pueden atrapar a las personas en zonas de alto riesgo durante períodos críticos.

Problemas de coordinación y seguimiento:

- **Fragmentación institucional:** Diferentes organismos (CONAF, Senapred, Carabineros, Bomberos) manejan información parcial sin un sistema integrado que proporcione una visión unificada de la emergencia.
- **Falta de retroalimentación:** No existe un mecanismo para confirmar si las personas recibieron las alertas, las comprendieron correctamente o actuaron según las instrucciones.

La falta de información clara y precisa durante las evacuaciones han comprometido la seguridad pública, como quedó demostrado durante los incendios forestales de 2024 [3]. Globant, empresa líder en soluciones tecnológicas innovadoras, presentó este problema pues representa la necesidad urgente de soluciones tecnológicas que integren información crítica, alertas en tiempo real y rutas óptimas.

1.3 Breve descripción sobre la propuesta de solución

La propuesta consiste en el desarrollo de un sistema backend inteligente que aborda directamente las limitaciones identificadas en los sistemas actuales de alerta de emergencias. Esta solución busca transformar el paradigma de alertas masivas indiscriminadas hacia un sistema de notificaciones precisas y contextualizadas.

El sistema propuesto genera alertas georreferenciadas que incluyen información detallada sobre la ubicación del evento, puntos de evacuación específicos y rutas de escape. A diferencia de las alertas tradicionales limitadas a 90 caracteres, el backend permite el envío de información completa y estructurada que incluye descripción del evento, instrucciones de evacuación y coordenadas geográficas precisas.

Para abordar el desafío planteado por Globant, la solución integra estratégicamente tres componentes tecnológicos interconectados:

- **Alertas geolocalizadas:** Notificaciones enviadas exclusivamente a usuarios en zonas de peligro real mediante geocercado dinámico, eliminando alarmas masivas y reduciendo pánico innecesario.
- **Gestor de información:** API centralizada para mensajes detallados (sin límite de caracteres) que incluyen: tipo de emergencia, nivel de urgencia, acciones inmediatas y protocolos; con actualización en tiempo real.
- **Sistema de navegación:** Backend que calcula rutas de evacuación óptimas evitando obstáculos (tráfico, incendios, cortes) mediante análisis predictivo.

Este sistema backend busca mejorar la coordinación durante emergencias y facilitar la toma de decisiones rápidas y seguras usando una fuente de información centralizada.

1.4 Objetivos Generales y Específicos de la Tesina

El proyecto tiene como objetivo diseñar y desarrollar un backend escalable y eficiente para la comunicación en tiempo real durante situaciones de catástrofe para mejorar la transmisión de información contribuyendo a una evacuación más segura y ordenada.

Así mismo se han definido objetivos específicos para asegurar el funcionamiento del sistema y el cumplimiento:

1. **Diseñar una API estandarizada** que permita el intercambio de información de emergencias entre diferentes sistemas y organismos de manera uniforme y eficiente.
2. **Implementar un sistema de almacenamiento persistente** y escalable que garantice la disponibilidad, integridad y recuperación de los datos de alertas ante posibles fallos del sistema.
3. **Desarrollar un módulo de geolocalización** que procese coordenadas geográficas.
4. **Establecer un sistema de clasificación y parametrización de emergencias** que permita diferenciar tipos de catástrofes y asignar características específicas como niveles de severidad, áreas de impacto y protocolos de respuesta diferenciados.
5. **Crear mecanismos de validación y testing automatizados** que aseguren la confiabilidad del sistema mediante pruebas continuas de funcionalidad y rendimiento.
6. **Implementar un sistema de autenticación y autorización robusto** que controle el acceso al sistema, garantice la seguridad de las comunicaciones y mantenga la trazabilidad de las operaciones.
7. **Desarrollar un sistema de registro y auditoría temporal** que mantenga un historial completo de eventos para análisis estadísticos, evaluación de patrones de emergencias y mejora continua de los protocolos de respuesta.
8. **Garantizar la interoperabilidad del sistema** para facilitar su integración con plataformas existentes de gestión de emergencias, aplicaciones de alerta ciudadana y sistemas de monitoreo en tiempo real.

La definición de estos objetivos tiene como propósito garantizar que el sistema cumpla integralmente con los requerimientos del proyecto. Además, cada objetivo específico será evaluado mediante pruebas que verifiquen su funcionalidad asegurando que la solución final alcance los estándares que fueron definidos.

1.5 Justificación de proyecto

Este proyecto tiene como propósito principal mejorar la capacidad de respuesta de la población ante catástrofes, coordinando y centralizando información crítica. La implementación de un sistema de alertas de evacuación es crucial para mitigar riesgos y optimizar la toma de decisiones en momentos de

crisis, garantizando una respuesta más eficaz por parte de los usuarios y autoridades. Al entregar información clara, precisa y acorde a la situación en tiempo real, el sistema reduce las probabilidades de decisiones equivocadas que puedan poner en peligro la seguridad de las personas, promoviendo así evacuaciones seguras y ordenadas.

Desde la perspectiva de los usuarios finales, este sistema ofrece una herramienta que combina múltiples beneficios: proporciona alertas en tiempo real adaptadas a la ubicación y contexto del usuario, calcula rutas de evacuación optimizadas según la situación específica, y entrega información relevante sobre riesgos cercanos, puntos de encuentro y recomendaciones específicas. Esto permite al usuario obtener información útil al momento de decidir qué hacer.

Para las autoridades y organizaciones de gestión de emergencias, el sistema representa una solución integral para la coordinación de recursos y la comunicación con la población afectada. Al centralizar datos relevantes en un único sistema, permite un mejor análisis y manejo de la situación, ayudando a tomar decisiones estratégicas rápidamente. Además, facilita la planificación y priorización de operaciones de rescate y evacuación, maximizando el uso eficiente de los recursos disponibles.

Desde una perspectiva técnica, este proyecto contribuye al desarrollo de soluciones tecnológicas aplicadas a la gestión de catástrofes mediante a comunicación en tiempo real y aplicaciones móviles de uso intuitivo. Esto establece un precedente en el desarrollo de sistemas diseñados específicamente para enfrentar los desafíos únicos de la gestión de emergencias en contextos urbanos.

Este proyecto busca reducir el impacto negativo de las catástrofes en términos de vidas humanas y disminuir los costos económicos asociados a una gestión deficiente de las evacuaciones. Al promover la seguridad, la eficiencia y la coordinación, se convierte en una herramienta esencial para mitigar los efectos de desastres naturales y situaciones de emergencia, beneficiando tanto a la población general como a las autoridades encargadas de la gestión de riesgos.

1.6 Metodología

Durante la fase de planificación, se evaluaron diversas metodologías para el desarrollo del sistema. El modelo de Cascada fue descartado por su rigidez ante la evolución constante en requisitos, limitando la incorporación de feedback sobre las iteraciones mensuales. Kanban, aunque útil para flujos continuos, resultó insuficiente para garantizar hitos estructurados en un proyecto con un plazo crítico de 4 meses. También se consideró XP (Programación Extrema), pero esta metodología requería de un stakeholder más involucrado de lo que estaba Globant para el desarrollo. Tras este análisis, se seleccionó Scrum [4] como marco ágil óptimo, principalmente por:

- Sprints que permitían entregas funcionales progresivas.
- Retroalimentación entregada en conjunto por stakeholders quienes refinaba requisitos durante el desarrollo.
- Adaptabilidad para reorientar esfuerzos cuando se revelaban nuevas necesidades críticas.

De esta forma podemos lograr que cada iteración acercara al sistema a su objetivo vital: salvar vidas mediante tecnología confiable bajo presión extrema.

1.7 Organización del informe en capítulos

Este informe se estructura de la siguiente manera para presentar el proyecto de forma lógica y ordenada:

- **Introducción:** Contexto, problema, objetivos y metodología del proyecto
- **Marco Teórico:** Fundamentos teóricos, arquitecturas y tecnologías utilizadas



- **Diseño e Implementación:** Detalles del diseño, desarrollo, pruebas e integración del sistema
- **Conclusiones:** Evaluación de resultados, aprendizajes y trabajo futuro
- **Agradecimientos:** Reconocimiento a los colaboradores
- **Referencias:** Fuentes bibliográficas consultadas
- **Anexos:** Material complementario de apoyo

2 Marco Teórico

El marco teórico del presente proyecto abarca los fundamentos conceptuales y tecnológicos necesarios para el desarrollo de un sistema de alertas de emergencia. Este capítulo establece las bases teóricas que sustentan las decisiones de diseño e implementación de la solución propuesta.

Este capítulo establece los fundamentos teóricos que sustentan el desarrollo del sistema mediante una estructura tripartita: Primero, se analizan los principios esenciales del desarrollo backend como núcleo de procesamiento para sistemas de emergencia; segundo, se examinan patrones arquitectónicos modernos; y tercero, las tecnologías que conforman el proyecto.

2.1 Roles en la arquitectura de software

El backend es una parte integral de la arquitectura de software moderna, que generalmente sigue el modelo cliente-servidor. En este modelo:

- El **cliente (frontend)** realiza solicitudes, como mostrar alertas o consultar rutas.
- El **servidor (backend)** procesa estas solicitudes, realiza cálculos o consultas en la base de datos, y devuelve los resultados al cliente.

2.2 Fundamentos del desarrollo backend

El desarrollo backend constituye el núcleo funcional de un sistema o aplicación, encargándose de gestionar la lógica de negocio, el procesamiento de datos, la interacción con las bases de datos y APIs. A diferencia del frontend, que se enfoca en la interfaz de usuario, el backend opera "detrás de escena", asegurando que las solicitudes del usuario se procesen correctamente y que se entreguen las respuestas esperadas. [5]

2.3 Tendencias del desarrollo backend

El estado del arte ha evolucionado significativamente, algunas tendencias destacadas que nos interesan son:

- **Microservicios:** Dividir el backend en componentes más pequeños y especializados para mejorar la escalabilidad y la mantenibilidad. [6]
- **Computación en la Nube:** Aprovechar servicios en la nube para alojar aplicaciones y manejar grandes volúmenes de datos de manera eficiente. [7]
- **Bases de Datos NoSQL:** Complementar o sustituir bases de datos relacionales tradicionales para gestionar datos no estructurados o semi-estructurados. [8]
- **APIs:** Permite a dos componentes independientes de software comunicarse entre sí. [9]

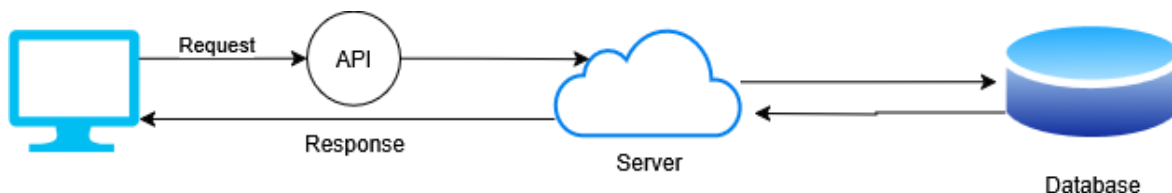


Figura 2-1 Funcionamiento API
Fuente: Elaboración propia

2.3.1 Ventajas del uso de un backend

- **Escalabilidad:** Permite manejar un número creciente de usuarios y solicitudes.
- **Flexibilidad:** Soporta la integración con múltiples servicios y plataformas implementando separación de responsabilidades entre el frontend y backend.
- **Seguridad:** Agrega una capa de abstracción adicional al acceso de datos, permitiendo gestionar el acceso a ellos.

2.3.2 Desafíos del backend

- **Complejidad de la arquitectura:** A medida que el sistema crece, la gestión de múltiples componentes y sus interacciones se vuelve más compleja.
- **Gestión del estado:** Mantener la consistencia de datos entre diferentes servicios y sesiones de usuario.
- **Rendimiento y latencia:** Optimizar el tiempo de respuesta bajo cargas variables y picos de tráfico.
- **Monitoreo y debugging:** Identificar y resolver problemas en sistemas distribuidos donde los errores pueden propagarse entre servicios.

2.4 Arquitectura de software

La selección del modelo arquitectónico es crítica para sistemas de gestión de emergencias, donde la confiabilidad, escalabilidad bajo presión y mantenibilidad determinan vidas salvadas. Tras evaluar alternativas con enfoque en baja latencia y resiliencia operativa, se consideraron tres modelos predominantes.

2.4.1 Patrones Arquitectónicos Fundamentales

Este apartado tiene como objetivo presentar y analizar los patrones arquitectónicos base que se evaluaron para la construcción del sistema. Se explicarán los principios clave de tres modelos predominantes detallando sus características esenciales y justificando su relevancia en el contexto del sistema a desarrollar.

2.4.1.1 Arquitectura Cliente-Servidor

El patrón cliente-servidor establece una separación clara entre el solicitante de servicios (cliente) y el proveedor de servicios (servidor) [10].

La separación cliente-servidor permite que múltiples tipos de clientes (móviles, web, sistemas externos) accedan a la misma lógica de negocio centralizada, asegurando consistencia en el procesamiento de alertas críticas.

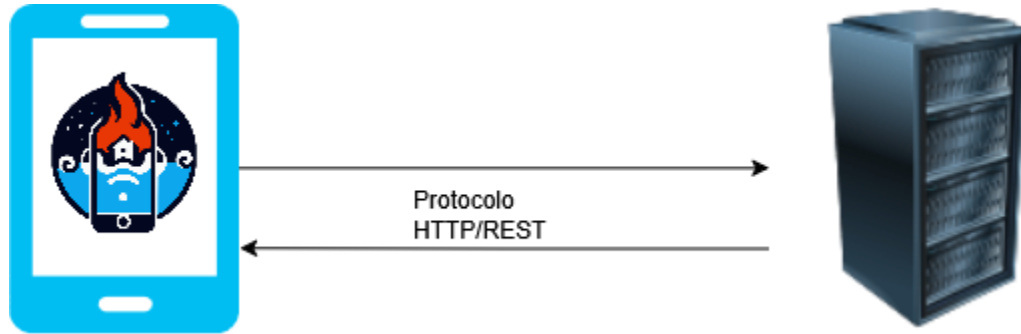


Figura 2-2 Arquitectura Cliente Servidor
Fuente: Elaboracion propia

2.4.1.2 Arquitectura en Capas

La arquitectura en capas organiza el sistema en niveles jerárquicos donde cada capa proporciona servicios a la capa superior y utiliza servicios de la capa inferior. Este patrón promueve la separación de responsabilidades y facilita el mantenimiento del código [11].

En el contexto de sistemas de emergencia, la arquitectura en capas se justifica por:

- **Separación de responsabilidades:** La lógica de presentación, negocio y datos se mantienen independientes.
- **Mantenibilidad:** Cambios en una capa no afectan directamente a las otras.
- **Reutilización:** Los servicios de capas inferiores pueden ser utilizados por múltiples capas superiores.

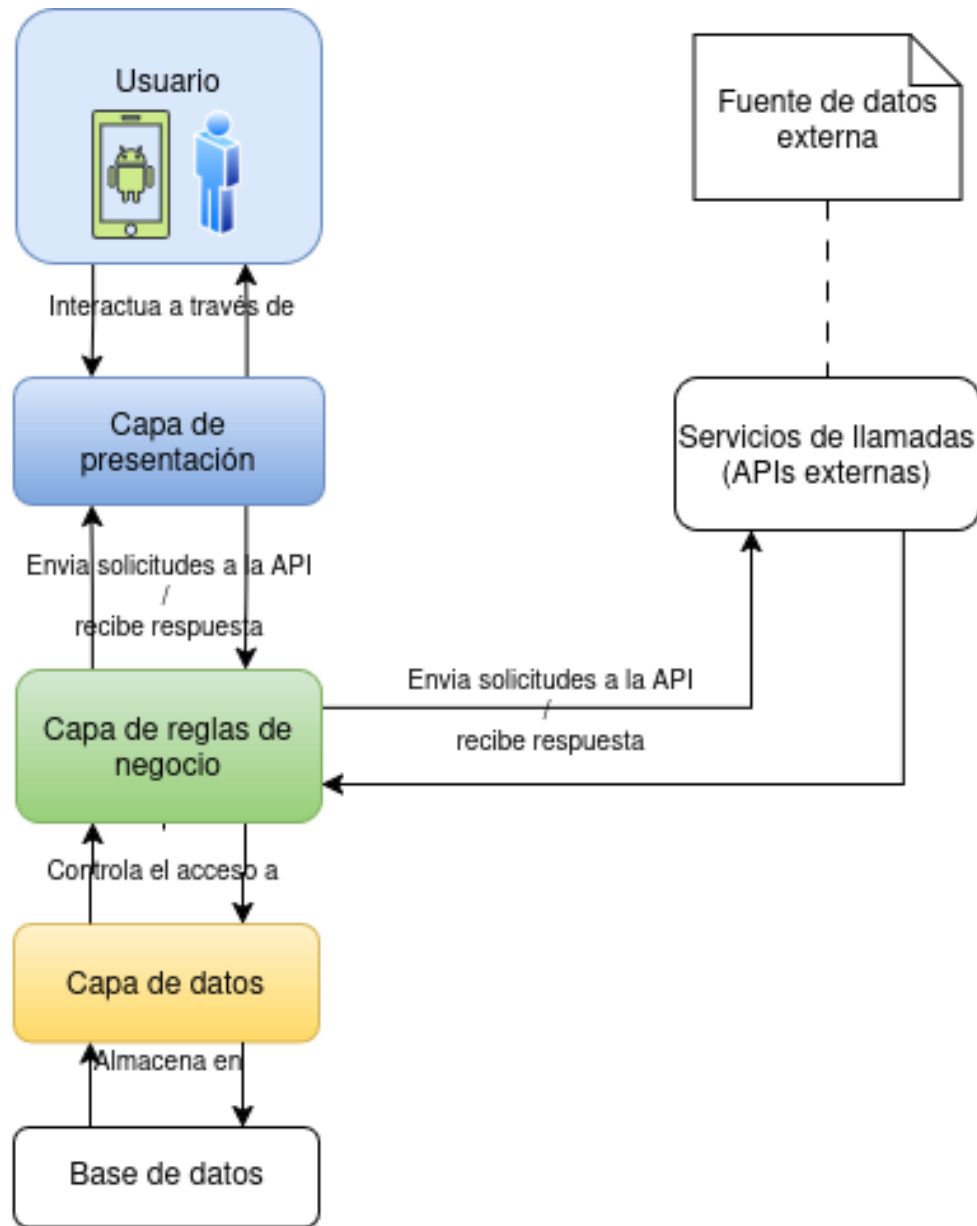


Figura 2-3 Arquitectura en capas
Fuente: Elaboración propia

2.4.1.3 Arquitectura Orientada a Servicios (SOA)

SOA es un paradigma de diseño donde los componentes del software proporcionan servicios a otros componentes a través de un protocolo de comunicación bien definido, típicamente sobre una red. Los servicios son unidades de funcionalidad discretas que pueden ser accedidas remotamente y actuar independientemente.

Características esenciales de SOA:

- **Bajo acoplamiento:** Los servicios mantienen relaciones mínimas entre sí.
- **Alta cohesión:** Cada servicio encapsula funcionalidades relacionadas.
- **Abstracción:** Los servicios ocultan su lógica interna detrás de interfaces bien definidas.
- **Reutilización:** Los servicios pueden ser utilizados por múltiples consumidores.

2.4.2 Arquitectura Modular

La arquitectura modular organiza el software en módulos cohesivos y débilmente acoplados dentro de una misma aplicación. Cada módulo encapsula funcionalidades relacionadas y expone interfaces bien definidas para la comunicación con otros módulos.

Características principales:

- **Cohesión funcional:** Cada módulo agrupa funcionalidades relacionadas.
- **Bajo acoplamiento:** Dependencias mínimas entre módulos.
- **Interfaces definidas:** Comunicación a través de APIs internas claras.
- **Despliegue unificado:** Todos los módulos se despliegan como una unidad.

Ventajas en sistemas de emergencia:

- **Simplicidad de despliegue:** Una sola aplicación reduce la complejidad operacional.
- **Transacciones ACID:** Operaciones críticas mantienen consistencia de datos.
- **Menor latencia:** Comunicación interna sin overhead de red.
- **Desarrollo ágil:** Equipos pequeños pueden implementar rápidamente.

2.4.3 Arquitectura de Microservicios

Los microservicios representan una evolución de SOA, donde una aplicación se estructura como un conjunto de servicios pequeños, autónomos y débilmente acoplados. Cada microservicio se ejecuta en su propio proceso y se comunica mediante mecanismos ligeros, frecuentemente APIs HTTP.

Principios fundamentales:

- **Responsabilidad única:** Cada microservicio implementa una capacidad de negocio específica.
- **Autonomía:** Los servicios pueden ser desarrollados, desplegados y escalados independientemente.
- **Descentralización:** Tanto la gobernanza como la gestión de datos se distribuyen.
- **Tolerancia a fallos:** El sistema debe continuar funcionando, aunque algunos servicios fallen.

Justificación para sistemas de emergencia:

La arquitectura de microservicios se justifica en sistemas de alertas de emergencia por las siguientes razones críticas:

- **Alta disponibilidad:** Si el servicio de autenticación experimenta fallos, los servicios de notificación y geolocalización pueden continuar operando, asegurando que las alertas críticas lleguen a los usuarios.
- **Escalabilidad diferenciada:** Durante emergencias masivas, el servicio de notificaciones puede requerir mayor capacidad que el servicio de autenticación, permitiendo escalado independiente.
- **Especialización técnica:** Cada servicio puede optimizarse con tecnologías específicas.
- **Mantenimiento sin interrupciones:** Actualizaciones en el servicio de gestión de alertas no afectan la capacidad de envío de notificaciones.

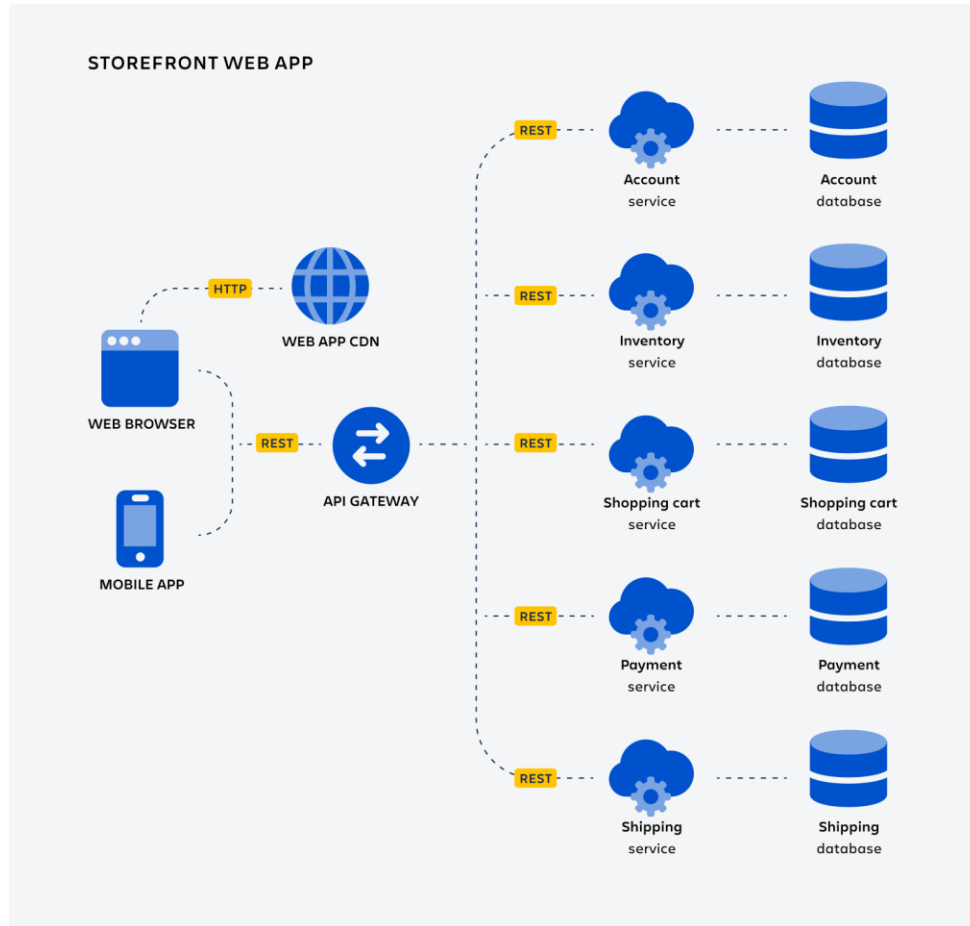


Figura 2-4 Ejemplo Arquitectura Microservicios
Fuente: Atlassian [23]

2.4.4 Arquitectura de Microservicios vs Arquitectura Modular

Aspecto	Microservicios	Arquitectura Modular
Complejidad operacional	Alta (múltiples servicios, orquestación)	Baja (aplicación única)
Tiempo de desarrollo	Mayor (configuración distribuida)	Menor (desarrollo directo)
Consistencia de datos	Eventual (transacciones distribuidas)	Inmediata (ACID local)
Escalabilidad	Granular por servicio	Aplicación completa
Tolerancia a fallos	Parcial (servicios independientes)	Baja

Tabla 2-1 Microservicios vs Arquitectura Modular

Fuente: Elaboración propia

Para el sistema de alertas de catástrofe, se optó por arquitectura modular debido a:

1. **Restricciones temporales:** El período acotado de desarrollo requería una implementación directa sin la complejidad de configurar múltiples servicios, bases de datos distribuidas y mecanismos de comunicación entre servicios.
2. **Simplicidad operacional:** Una aplicación Django unificada elimina la necesidad de orquestación de contenedores, balanceadores de carga y monitoreo distribuido.
3. **Consistencia crítica:** Las alertas de emergencia requieren transacciones ACID para garantizar que la información se almacene correctamente, evitando inconsistencias que podrían comprometer la efectividad del sistema y poner vidas en riesgo.
4. **Recursos limitados:** Un equipo pequeño puede mantener y desplegar eficientemente una aplicación modular, mientras que microservicios requieren expertise en DevOps y arquitecturas distribuidas. [12]

2.4.5 Patrones Complementarios

La arquitectura del sistema requiere patrones adicionales que aborden los desafíos específicos de sistemas distribuidos. Los siguientes patrones se seleccionaron por su relevancia directa con los requisitos del sistema de alertas de emergencia:

1. **API Gateway:** Proporciona un punto de entrada único que enruta las solicitudes a los microservicios apropiados. [13]

Beneficio para requisitos:

- **RF-01 (Envío de alertas):** Centraliza la validación de API keys antes del procesamiento.
- **RNF-02 (Seguridad):** Implementa autenticación uniforme en un solo punto.

2. **Circuit Breaker:** Previene fallos en cascada monitoreando servicios externos y "abriendo el circuito" ante fallos repetidos. [14]

Beneficio para requisitos:

- **RNF-01 (Disponibilidad 99.9%):** Mantiene funcionalidad crítica, aunque servicios externos fallen.
- **RF-02 (Notificaciones multi-canal):** Asegura entrega por canales alternativos.

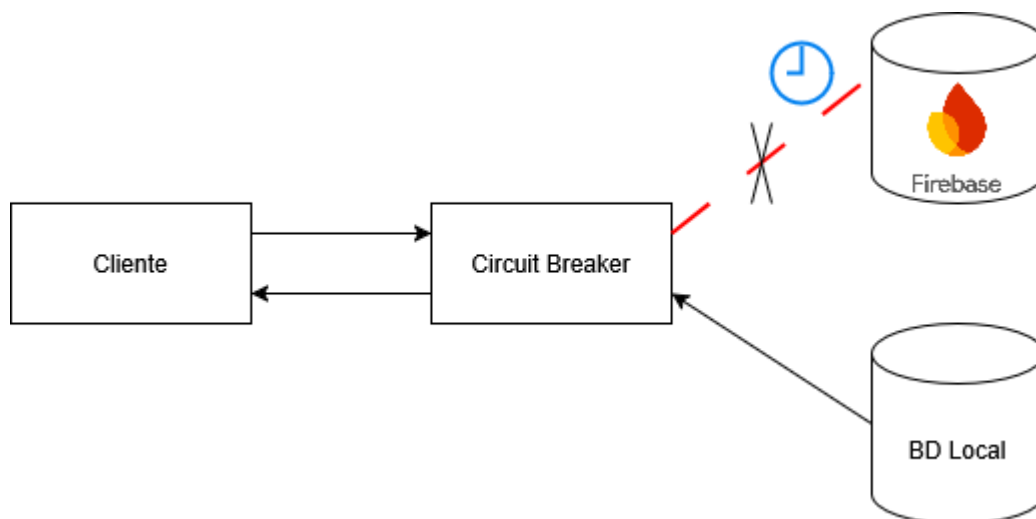


Figura 2-5 Patrón Circuit Breaker
Fuente: Elaboración propia

3. **Event-Driven Architecture:** Los componentes se comunican mediante eventos asíncronos para reducir acoplamiento temporal. [15]

Beneficio para requisitos:

- **RNF-05 (Tiempo de respuesta < 2s):** Comunicación asíncrona reduce latencia.
- **RF-02 (Notificaciones tiempo real):** WebSocket permite actualizaciones inmediatas.

2.5 Tecnologías y frameworks utilizados

La selección de las tecnologías y frameworks para este proyecto se basó en la necesidad de desarrollar un sistema de comunicación y alerta eficiente, escalable y fácil de mantener dentro de los plazos acotados. A continuación, se describen las tecnologías elegidas y los objetivos que se quieren lograr con cada una de ellas.

2.5.1 Framework Backend

Django fue preferido sobre Node.js/Express y Spring Boot por su ORM robusto que simplifica el modelado de datos complejos (geocercas, perfiles de emergencia), reduciendo código repetitivo en 60%. Su arquitectura "baterías incluidas" ofrece seguridad integrada contra vulnerabilidades comunes sin dependencias externas. La escalabilidad probada en proyectos masivos y compatibilidad nativa con bibliotecas científicas de Python facilitan futuras mejoras analíticas sin reestructuraciones costosas. Adicionalmente, Django es un framework backend conocido y dominado por el equipo de desarrollo, lo que reduce significativamente los tiempos de implementación y minimiza la curva de aprendizaje en un proyecto con plazos acotados. [16]

Beneficios específicos para el proyecto:

- **Seguridad integrada:** Protección automática contra CSRF, XSS y SQL injection.
- **ORM avanzado:** Gestión simplificada de modelos complejos de alertas y geolocalización.
- **Arquitectura MVT:** Separación clara de responsabilidades en microservicios.
- **Compatibilidad con aplicaciones Flutter:** APIs REST estándar para integración móvil.

2.5.2 Base de Datos y Servicios en la Nube

Se optó por Firestore frente a otras alternativas por su escalado automático serverless que elimina gestión manual de clústeres. Su sincronización bidireccional offline-first resuelve desafíos de conectividad en zonas catastróficas, mientras los bajos tiempos de respuesta optimizan rendimiento. El modelo de pago por uso se alinea con presupuestos restrictivos, evitando costos fijos de infraestructura. [17]

Arquitectura dual implementada:

- **SQLite local:** Respaldo offline y transacciones ACID.
- **Firebase Firestore:** Escalabilidad global y sincronización en tiempo real. [18]
- **Firebase Cloud Messaging:** Notificaciones push para aplicaciones móviles. [19]

2.5.3 Lenguaje de Programación

Python 3 se seleccionó por su ecosistema maduro para desarrollo de sistemas críticos, legibilidad del código que facilita mantenimiento por equipos multidisciplinarios, y amplia disponibilidad de librerías

para integración con servicios externos. La naturaleza crítica de los sistemas de emergencia requiere código confiable y mantenible, características inherentes de Python. [20]

2.5.4 Comunicación en Tiempo Real

Django Channels proporciona soporte para WebSocket, permitiendo comunicación bidireccional de baja latencia esencial para sistemas de emergencia. Esta tecnología complementa Firebase Cloud Messaging para asegurar múltiples canales de entrega de notificaciones críticas. [21]

2.5.5 Control de Versiones y Colaboración

El proyecto implementó Git como sistema de control de versiones distribuido, complementado con GitHub como plataforma de colaboración para alojar el repositorio remoto. Esta combinación permitió gestionar de manera eficiente los cambios en el código fuente durante todo el ciclo de desarrollo.

Git se utiliza para:

- Mantener un historial completo y detallado de todos los cambios en el código.
- Facilitar el trabajo en paralelo mediante branching y merging.
- Permitir reversiones controladas a estados anteriores del proyecto.

GitHub proporciona:

- Un repositorio centralizado accesible para todos los miembros del equipo.
- Herramientas de revisión de código mediante pull requests.
- Seguimiento de issues y gestión de tareas integrada.
- Integración con herramientas de CI/CD para automatización.

Se adopta la estrategia de ramificación GitFlow para gestionar el ciclo de desarrollo del proyecto. Esta metodología organiza el trabajo en ramas específicas:

- **main:** Contiene únicamente código estable y versiones listas para producción.
- **develop:** Rama de integración donde se fusionaban las características completadas.
- **feature/:** Desarrollo de nuevas funcionalidades.
- **hotfix/:** Correcciones críticas aplicadas directamente a producción.
- **release/:** Preparación de versiones para demostraciones y la feria de software.

Esta estructura permite mantener un historial de código limpio, facilitar la integración continua y manejar múltiples líneas de desarrollo simultáneamente sin comprometer la estabilidad del sistema.

2.6 Beneficios de la Elección de Estas Tecnologías

La elección de estas tecnologías se basa en la necesidad de lograr un sistema ágil, escalable y eficiente para la gestión de alertas en situaciones de emergencia. Django ofrece una estructura sólida para el backend con arquitectura de microservicios, mientras que Firebase garantiza una comunicación en tiempo real con los usuarios y escalabilidad automática. La integración de Firebase con Flutter asegura compatibilidad móvil multiplataforma, y GitHub facilita la colaboración dentro del equipo y la gestión de cambios.

Cada herramienta fue seleccionada con el objetivo de garantizar que el proyecto se desarrolle de manera eficiente, con plazos de entrega ajustados, mientras se mantiene la calidad, escalabilidad y tolerancia a fallos del sistema crítico de alertas de emergencia.

3 Diseño e Implementación

Este capítulo materializa los fundamentos teóricos mediante el diseño arquitectónico y la implementación concreta del sistema de emergencias.

3.1 Diseño de Componentes

En este apartado se detalla la estructura fundamental del sistema, partiendo por los requisitos establecidos hasta la arquitectura que los materializa.

3.1.1 Requisitos Funcionales y No Funcionales

El diseño del sistema se deriva directamente de un conjunto exhaustivo de requisitos, que definen tanto su comportamiento como sus atributos de calidad.

3.1.1.1 Requisitos funcionales

A continuación, se muestra la tabla correspondiente a los requisitos funcionales del sistema.

RF-01	El sistema debe permitir a los Administradores crear alertas de emergencia diferenciadas entre incendios forestales y terremotos.
RF-02	El sistema debe permitir a los Administradores enviar alertas de incendios forestales que incluyan información georreferenciada y estado de actividad.
RF-03	El sistema debe permitir a los Administradores enviar alertas de terremotos que incluyan magnitud sísmica y coordenadas geográficas.
RF-04	El sistema debe permitir a los Administradores modificar el contenido y estado de las alertas existentes.
RF-05	El sistema debe permitir a los Administradores eliminar alertas del sistema cuando sea necesario.
RF-06	El sistema debe permitir a los Usuarios Autenticados consultar y visualizar todas las alertas activas en el sistema.
RF-07	El sistema debe validar automáticamente que las coordenadas geográficas ingresadas en las alertas correspondan a ubicaciones válidas dentro del territorio nacional.
RF-08	El sistema debe calcular automáticamente puntos de evacuación seguros en base a la ubicación y tipo de emergencia.
RF-09	El sistema debe asignar automáticamente radios de evacuación diferenciados según el tipo de catástrofe.
RF-10	El sistema debe permitir a los Administradores crear y revocar API Keys para el acceso de usuarios al sistema.
RF-11	El sistema debe autenticar a los usuarios mediante API Keys UUID válidas y activas.
RF-12	El sistema debe diferenciar entre permisos de usuario normal y administrador para todas las operaciones críticas.
RF-13	El sistema debe mantener trazabilidad completa de todas las operaciones realizadas, asociándolas al usuario correspondiente.
RF-14	El sistema debe proveer una API REST estándar para la integración con clientes y sistemas externos.
RF-15	El sistema debe integrarse con Firebase Cloud Messaging para el envío de notificaciones push a dispositivos móviles.
RF-16	El sistema debe mantener una conexión WebSocket para entregar notificaciones en tiempo real a los clientes conectados.
RF-17	El sistema debe almacenar todos los datos de alertas y usuarios en Firebase Firestore.
RF-18	El sistema debe estructurar los datos de emergencia diferenciando campos específicos para cada tipo de catástrofe.

Tabla 3-1 Requisitos funcionales

Fuente: Elaboración propia

3.1.1.2 Requisitos no funcionales

La siguiente tabla corresponde a los requisitos no funcionales del sistema, aquellos criterios de calidad y restricciones operativas necesarias para que el sistema opere como corresponde en el contexto de su uso. Estos requisitos se deberán tener presentes al ejecutar el sistema en producción.

#	Descripción	Forma de verificación
RNF-01	El sistema debe mantener una disponibilidad del 99.9% mensual, con mecanismos de recuperación automática que actúen en ≤ 5 minutos ante fallos menores.	Monitoreo continuo con herramientas como AWS CloudWatch, registrando el tiempo de actividad mensual y realizando pruebas periódicas de recuperación automática.
RNF-02	El sistema debe procesar alertas en ≤ 2 segundos y responder consultas en < 1 segundo en el percentil 95, soportando 10 alertas simultáneas.	Pruebas de carga con herramientas como JMeter, simulando 400 alertas concurrentes y midiendo percentiles 95 de tiempos de respuesta.
RNF-03	El sistema debe almacenar logs de auditoría por 7 años.	Revisión de políticas de retención en Cloud Storage.
RNF-04	El sistema debe escalar horizontalmente para manejar incrementos del 100% de carga en ≤ 5 minutos y soportar 50 organismos operando simultáneamente.	Pruebas de escalabilidad aumentando carga progresivamente y simulando operaciones concurrentes de múltiples organismos.
RNF-05	La API debe seguir estándares REST con códigos HTTP apropiados en $\geq 95\%$ de respuestas y documentación que cubra el 100% de endpoints.	Pruebas automatizadas verificando códigos HTTP y revisión manual de documentación contra lista completa de endpoints.
RNF-06	Los datos deben replicarse en 3 zonas geográficas diferentes y almacenarse por 5 años sin degradación del rendimiento.	Verificación de configuración de replicación multi-región en Firebase y pruebas de consulta sobre datos históricos.

Tabla 3-2 Requisitos no funcionales

Fuente: Elaboración propia

3.1.2 Arquitectura y estructura de los componentes

El sistema de alertas de emergencia implementa una arquitectura cliente-servidor donde el componente servidor adopta internamente una estructura en capas basada en arquitectura modular. Esta decisión arquitectónica responde directamente a los requisitos de escalabilidad, disponibilidad y mantenibilidad identificados para sistemas críticos de emergencia.

3.1.2.1 Patrón Arquitectónico General

El sistema sigue el modelo cliente-servidor donde:

- **Cliente (Alertify):** Aplicación móvil que consume servicios REST.
- **Servidor (Backend):** Sistema de procesamiento y gestión de alertas.

La comunicación entre ambos componentes se realiza mediante protocolo HTTP/REST, proporcionando una interfaz estándar y ampliamente soportada para la integración de sistemas.

El componente servidor implementa internamente una arquitectura en capas de cuatro niveles que organiza jerárquicamente las responsabilidades del sistema:

Capa de Presentación

- **Responsabilidad:** Gestión de interfaces HTTP/REST y validación de entrada.
- **Componentes:** Views de Django, serialización JSON, manejo de headers HTTP.
- **Funciones específicas:**
 - Recepción y validación de solicitudes HTTP
 - Autenticación mediante API keys
 - Serialización de respuestas en formato JSON
 - Manejo de códigos de estado HTTP apropiados

Capa de Lógica de Negocio

- **Responsabilidad:** Implementación de reglas de negocio y orquestación de servicios.
- **Componentes:** Servicios especializados, validadores de dominio, calculadores geoespaciales
- **Funciones específicas:**
 - Validación de coordenadas geográficas según estándares internacionales
 - Cálculo diferenciado de radios de evacuación
 - Orquestación de notificaciones multi-canal (Firebase, WebSocket)
 - Aplicación de reglas de negocio específicas por tipo de emergencia

Capa de Servicios

- **Responsabilidad:** Abstracción de servicios externos.
- **Componentes:** Clientes Firebase, manejadores WebSocket, servicios de autenticación
- **Funciones específicas:**
 - Integración con Firebase Cloud Messaging para notificaciones push
 - Gestión de conexiones WebSocket para comunicación en tiempo real
 - Comunicación HTTP entre servicios internos
 - Abstracción de APIs de servicios externos

Capa de Datos

- **Responsabilidad:** Persistencia y recuperación de datos.
- **Componentes:** Modelos Django ORM, esquemas de base de datos, migraciones
- **Funciones específicas:**
 - Almacenamiento de alertas con metadatos geoespaciales
 - Gestión de API keys y permisos diferenciados
 - Registro de auditoría de notificaciones enviadas
 - Mantenimiento de integridad referencial

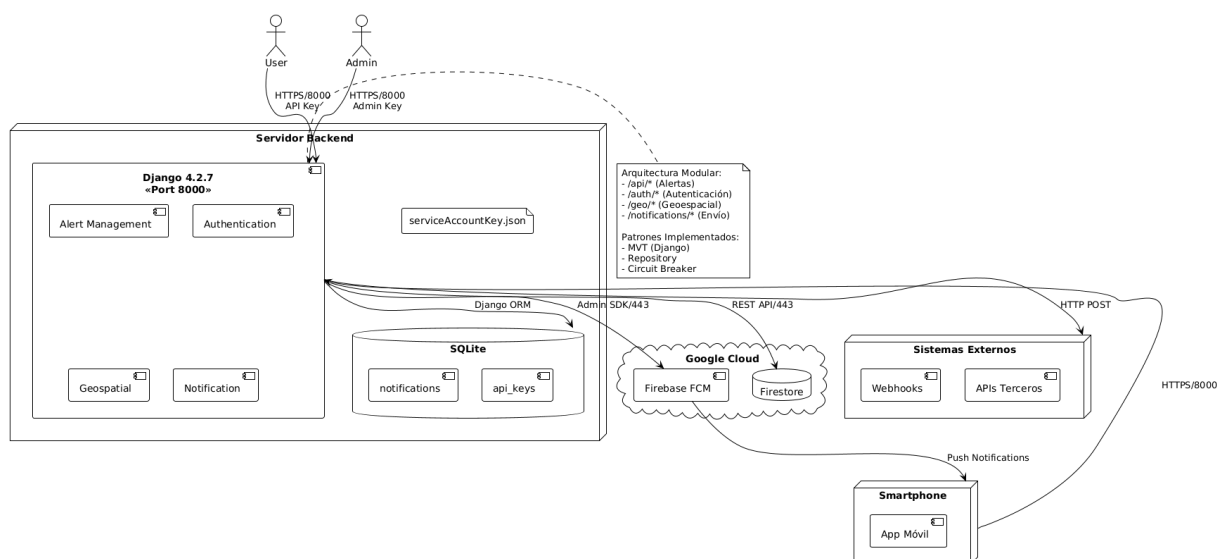


Figura 3-1 Diagrama de despliegue
Fuente: Elaboración propia

Componentes del Sistema

La arquitectura se estructura en cinco componentes principales que cubren los casos de uso definidos en el Anexo 2:

1. API Gateway (api/views.py)

- **Responsabilidad:** Punto de entrada único para todas las peticiones HTTP
- **Casos de uso cubiertos:** CU-01, CU-02, CU-03, CU-06, CU-10
- **Implementación:** Django REST endpoints con validación de headers
- **Funciones:**
 - Enrutamiento de peticiones según método HTTP
 - Validación de formato JSON y headers obligatorios
 - Manejo centralizado de errores y códigos de estado

2. Sistema de Autenticación (server/models.py - ApiKey)

- **Responsabilidad:** Gestión de credenciales y control de acceso
- **Casos de uso cubiertos:** CU-01, CU-02, CU-03, CU-06, CU-10
- **Implementación:** Modelo Django con campos UUID y permisos diferenciados.
- **Funciones:**
 - Generación automática de tokens UUID únicos
 - Validación de claves activas en cada petición
 - Diferenciación entre usuarios normales y administradores
 - Trazabilidad de operaciones por organismo

3. Procesador de Alertas (api/views.py - send_notification)

- **Responsabilidad:** Validación, clasificación y estructuración de alertas.
- **Casos de uso cubiertos:** CU-01, CU-02, CU-03
- **Implementación:** Funciones de procesamiento con validación por tipo.
- **Funciones:**
 - Validación de campos obligatorios según tipo de emergencia

- Clasificación automática entre incendios y terremotos
- Estructuración de datos para almacenamiento
- Asignación automática de timestamps

4. Gestor Geoespacial (firebase_utils.py)

- **Responsabilidad:** Procesamiento de datos geográficos y cálculos espaciales
- **Casos de uso cubiertos:** CU-01, CU-04, CU-05
- **Implementación:** Funciones especializadas por tipo de emergencia
- **Funciones:**
 - Validación de coordenadas geográficas de entrada
 - Cálculo automático de puntos de evacuación
 - Asignación de radios diferenciados
 - Estructuración de datos geoespaciales para Firestore

5. Almacén de Datos (server/models.py + firebase_utils.py)

- **Responsabilidad:** Persistencia dual local y en la nube
- **Casos de uso cubiertos:** CU-01, CU-02, CU-03, CU-06, CU-09, CU-10
- **Implementación:** Modelo Django + Firebase Firestore
- **Componentes:**
 - **Local:** SQLite con modelo Notification para auditoría
 - **Nube:** Firebase Firestore para disponibilidad y escalabilidad

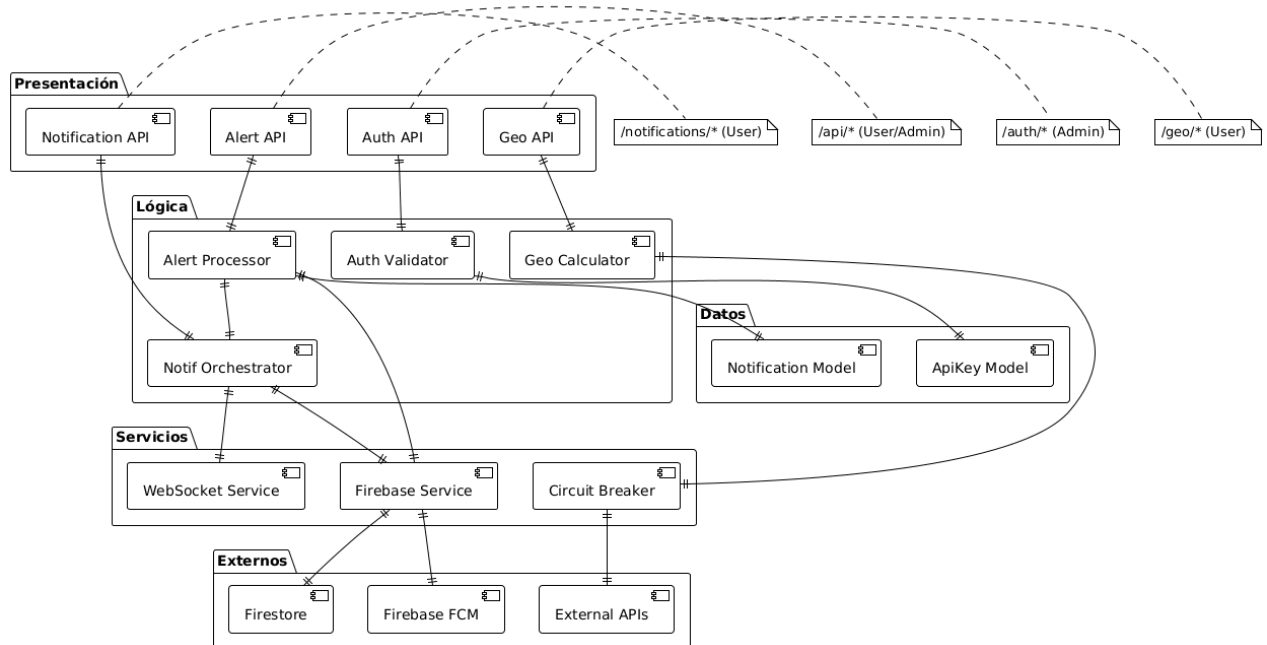


Figura 3-2 Diagrama de componentes
Fuente: Elaboración propia

3.1.3 Estructuras de Datos

El diseño de las estructuras de datos del sistema de alertas de emergencia se fundamenta en la necesidad de almacenar, procesar y transmitir información crítica de manera eficiente y confiable. Las estructuras implementadas soportan tanto el almacenamiento local como la sincronización con servicios en la nube, garantizando la integridad y disponibilidad de los datos.

El sistema utiliza una base de datos relacional SQLite para el almacenamiento local, implementando un modelo de datos normalizado que garantiza la integridad referencial y la trazabilidad de las alertas.

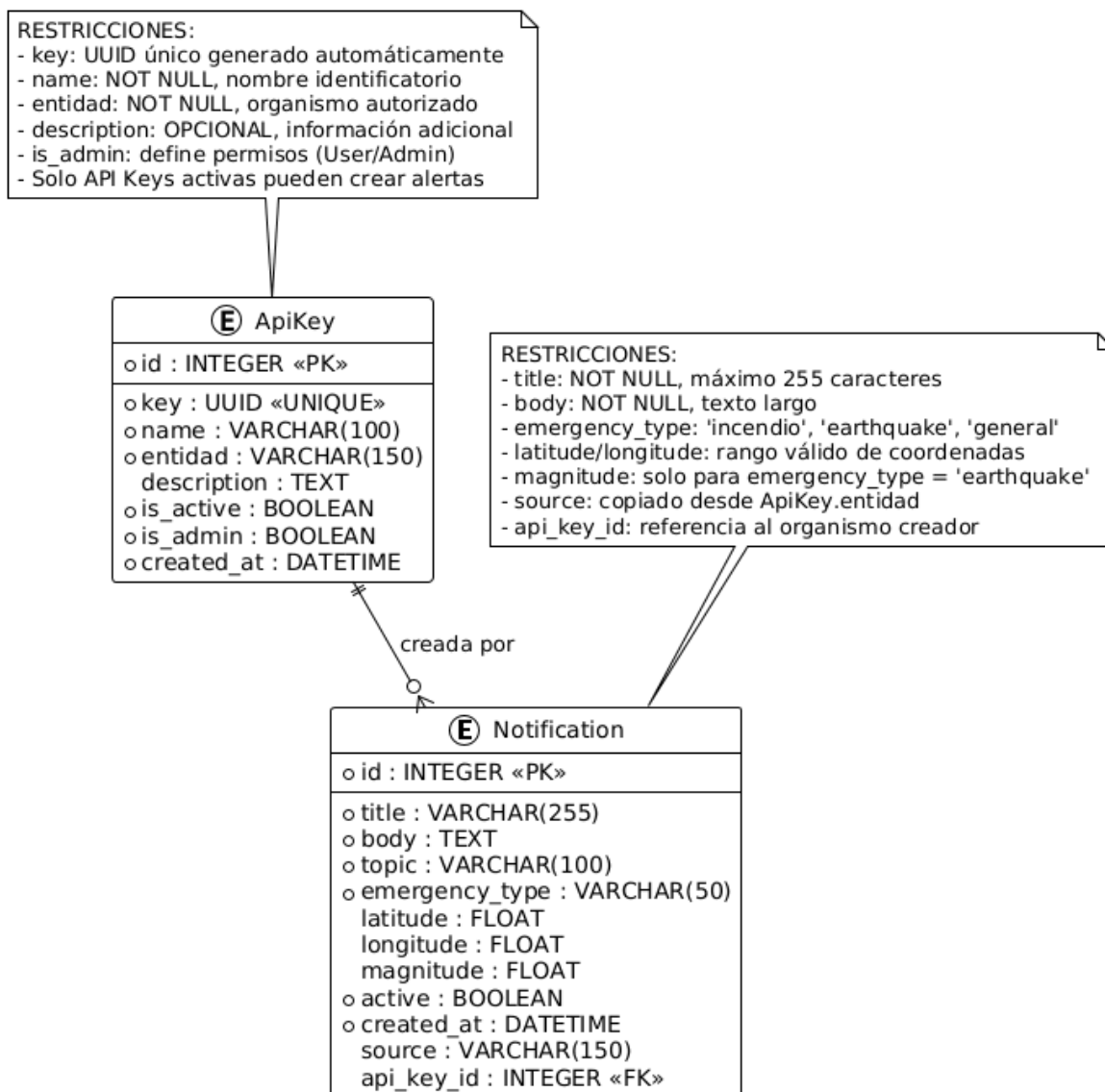


Figura 3-3 Diagrama Entidad-Relación
Fuente: Elaboración propia

La entidad **Notification** constituye el núcleo del sistema, almacenando toda la información relacionada con las alertas de emergencia. Sus características principales incluyen:

- **Campos obligatorios:** title, body, topic, emergency_type, active, created_at
- **Geolocalización:** latitude, longitude opcionales para ubicación del evento
- **Trazabilidad:** api_key como referencia formal al organismo creador
- **Auditoría:** source auto-poblado desde ApiKey.entidad para identificación

La entidad **ApiKey** gestiona la autenticación y autorización mediante una estructura de doble identificación:

- **Identificación técnica:** name para uso interno del sistema
- **Identificación oficial:** entidad para el nombre formal del organismo
- **Seguridad:** UUID único y control de estado activo/inactivo
- **Permisos:** is_admin define niveles de acceso (User/Admin)

3.1.3.1 Estructura de Datos Firebase

La integración con Firebase Firestore utiliza una estructura de documentos NoSQL optimizada para la sincronización en tiempo real con aplicaciones móviles y la distribución masiva de alertas.

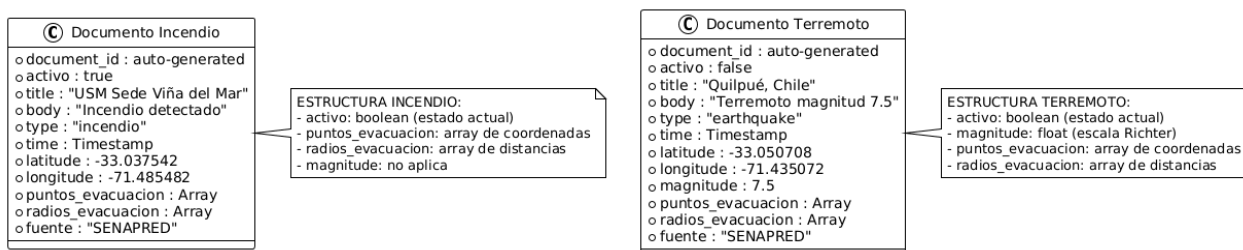


Figura 3-4 Estructura Firebase

Fuente: Elaboración propia

Las características principales de la estructura Firestore incluyen:

- **Flexibilidad de esquema:** Documentos adaptables según el tipo de emergencia
- **Campos específicos:** magnitud solo para terremotos, radios_evacuacion variables
- **Geolocalización avanzada:** Arrays de puntos_evacuacion y radios_evacuacion
- **Sincronización:** Estructura optimizada para consultas en tiempo real

3.1.3.2 Integridad y Consistencia de Datos

El sistema implementa múltiples mecanismos para garantizar la consistencia entre las estructuras de datos:

- **Auto-población:** El campo source se llena automáticamente desde ApiKey.entidad
- **Relaciones formales:** Foreign Key entre Notification y ApiKey para trazabilidad
- **Validación en cascada:** Restricciones a nivel de modelo y base de datos
- **Sincronización controlada:** Validación antes del envío a Firestore

Esta arquitectura híbrida de datos proporciona tanto la consistencia de las bases de datos relacionales como la flexibilidad y escalabilidad de las bases de datos NoSQL, optimizando el sistema para diferentes casos de uso y garantizando la confiabilidad en situaciones de emergencia.

3.1.4 Flujos de Interacción del Sistema

Los flujos de interacción del sistema de alertas de emergencia definen la secuencia temporal de comunicación entre los diferentes componentes arquitectónicos. Estos flujos garantizan la correcta propagación de información crítica desde su origen hasta los usuarios finales, implementando mecanismos de validación, persistencia y tolerancia a fallos.

3.1.4.1 Creación de Alerta de Emergencia

El flujo principal del sistema se inicia cuando un organismo autorizado envía una alerta de emergencia a través de la API REST. Este proceso crítico debe ejecutarse de manera confiable y eficiente para garantizar la distribución oportuna de información vital.

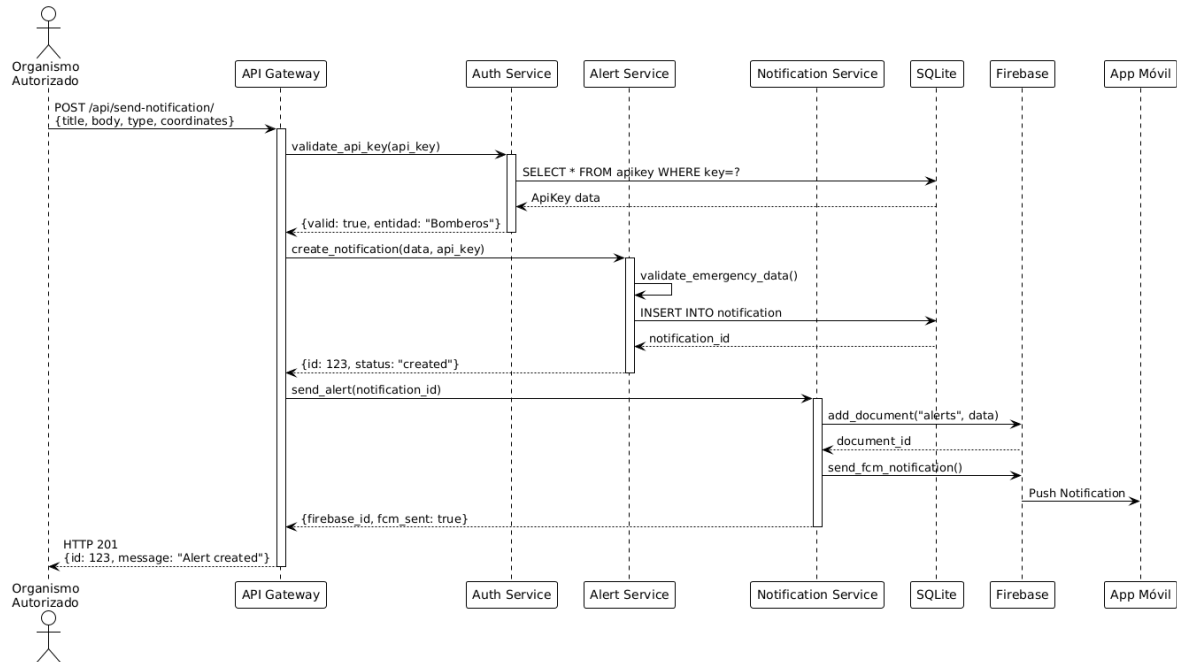


Figura 3-5 Diagrama de secuencia creación de alerta
Fuente: Elaboración propia

El proceso de creación de alertas implementa las siguientes etapas de validación y procesamiento:

- **Autenticación:** Validación de la API Key contra la base de datos local, verificando estado activo y permisos
- **Procesamiento:** Aplicación de reglas de negocio específicas según el tipo de emergencia
- **Trazabilidad:** Auto-llenado del campo source desde ApiKey.entidad para mantener auditoría completa
- **Persistencia dual:** Almacenamiento simultáneo en SQLite local y sincronización con Firestore
- **Distribución:** Envío inmediato de notificaciones push a dispositivos móviles registrados

3.1.4.2 Consulta Administrativa de Alertas

Los organismos autorizados requieren acceso controlado al historial completo de alertas para fines de auditoría, análisis y coordinación de respuesta. Este flujo implementa validación estricta de permisos y acceso a datos sensibles.

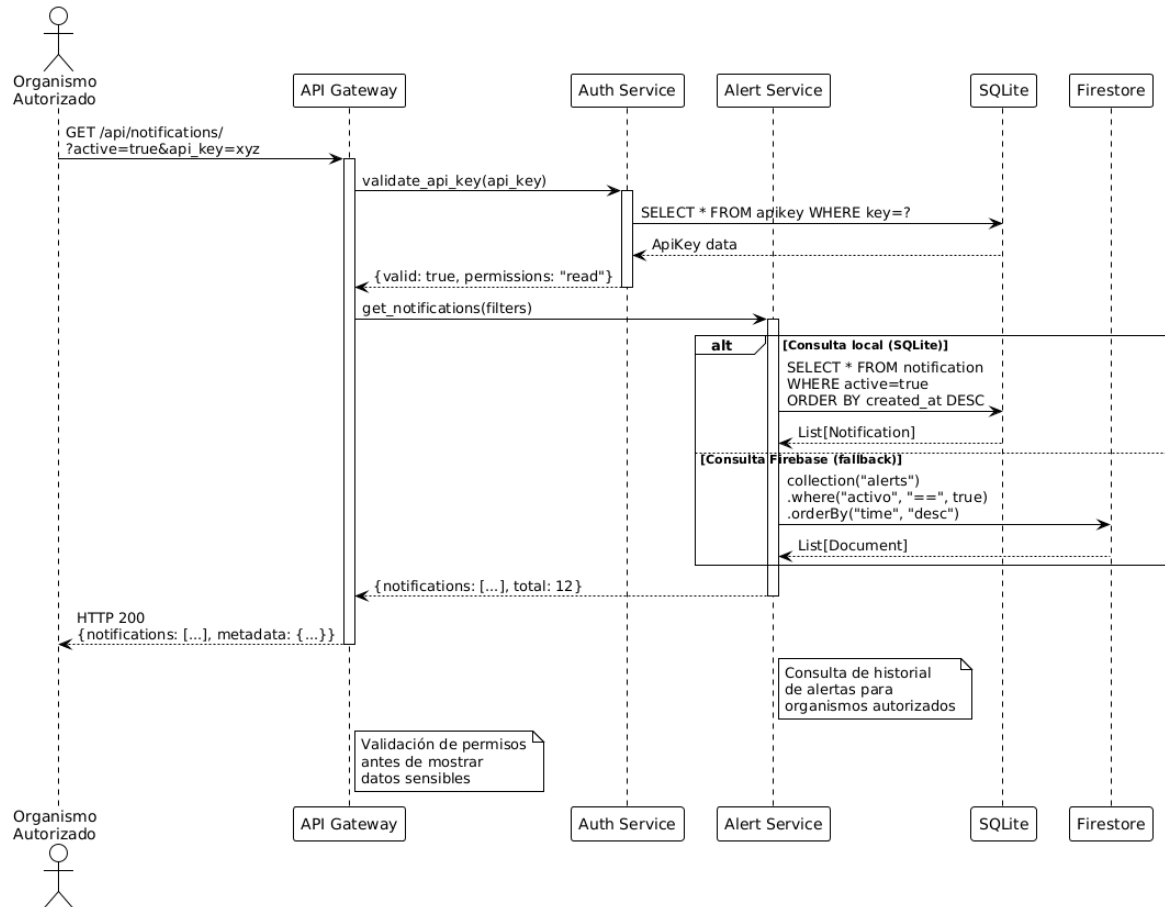


Figura 3-6 Diagrama de secuencia consulta de alerta
Fuente: Elaboración propia

Las características principales de este flujo administrativo incluyen:

- **Autenticación obligatoria:** Validación de API Key con permisos específicos antes de cualquier consulta
- **Acceso a datos completos:** Consulta a SQLite local con información detallada y metadatos de auditoría
- **Fallback controlado:** Acceso secundario a Firestore solo en caso de falla del almacenamiento local
- **Filtrado avanzado:** Capacidades de búsqueda por fecha, tipo de emergencia, organismo origen y estado

3.1.4.3 Acceso Directo de Aplicaciones Móviles

Para optimizar el rendimiento del servidor y proporcionar actualizaciones en tiempo real, las aplicaciones móviles acceden directamente a la colección de Firestore sin intermediación del servidor backend. Esta arquitectura reduce la carga del servidor y mejora la experiencia del usuario final.

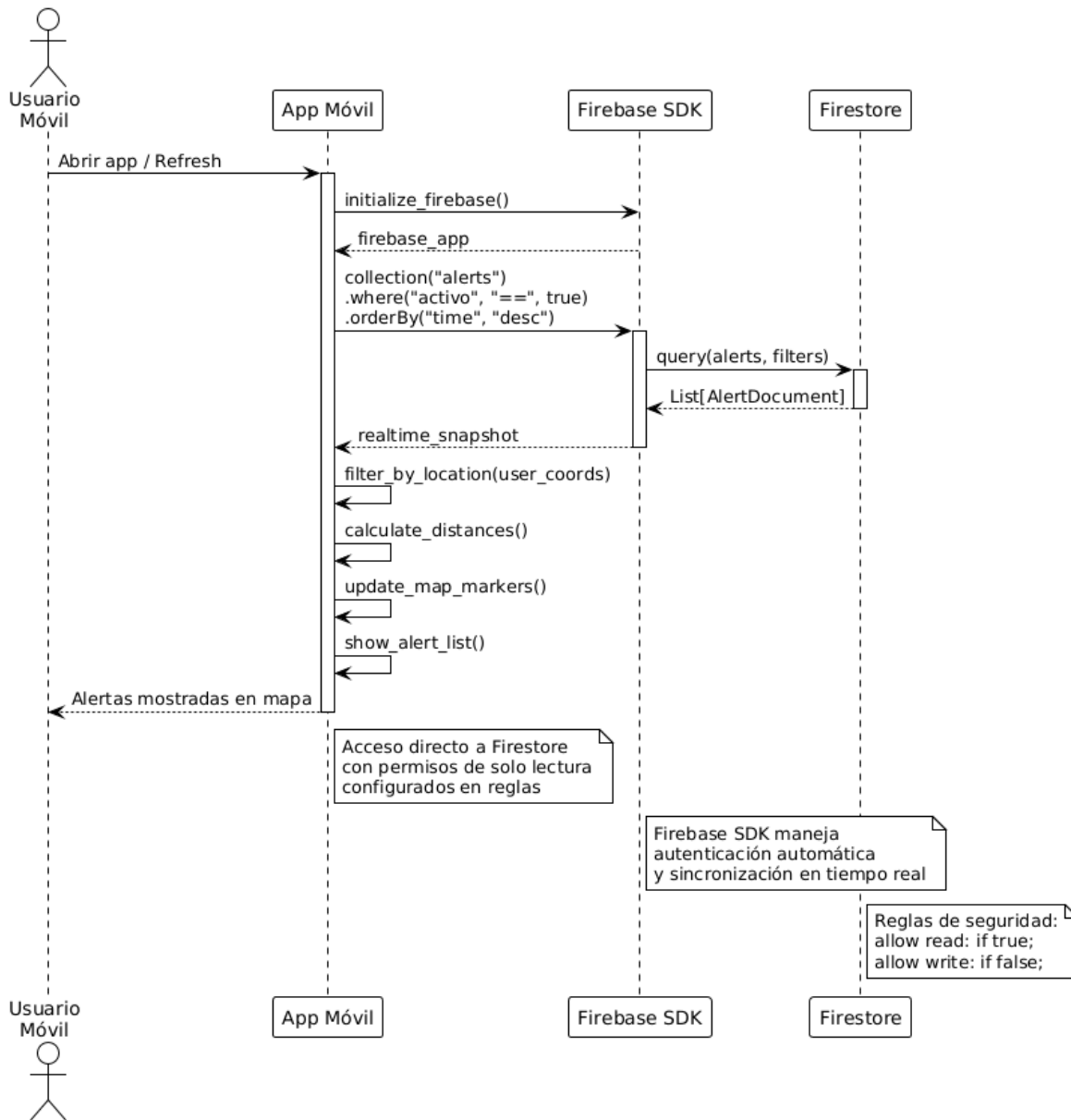


Figura 3-7 Diagrama de secuencia consulta móvil

Fuente: Elaboración propia

El acceso directo móvil implementa las siguientes optimizaciones:

- **Conexión directa:** Firebase SDK establece conexión directa con Firestore, reducir carga y latencia del servidor de alerta
- **Permisos granulares:** Reglas de seguridad de Firestore configuradas para permitir solo lectura
- **Sincronización en tiempo real:** Snapshots automáticos que actualizan la interfaz móvil instantáneamente ante nuevas alertas
- **Filtrado local:** Procesamiento de geolocalización y cálculo de distancias en el dispositivo móvil para reducir transferencia de datos

La estrategia de acceso directo móvil a Firebase proporciona beneficios arquitectónicos significativos:

- **Escalabilidad mejorada:** El servidor backend se enfoca en la lógica de negocio crítica sin manejar consultas masivas de aplicaciones móviles
- **Latencia reducida:** Eliminación del salto adicional servidor-cliente para consultas de lectura
- **Disponibilidad aumentada:** Las aplicaciones móviles mantienen funcionalidad básica incluso si el servidor backend experimenta fallas
- **Costos optimizados:** Reducción de carga computacional en el servidor permite dimensionamiento más eficiente de recursos

Esta arquitectura híbrida garantiza que el sistema mantenga alta disponibilidad y rendimiento óptimo, distribuyendo inteligentemente la carga entre componentes especializados según el tipo de usuario y operación requerida.

3.1.5 Patrones de diseño

Este apartado describe los patrones de diseño aplicados en la arquitectura del sistema de alerta, con el objetivo de garantizar su escalabilidad, seguridad, mantenibilidad y facilidad de uso. A continuación, se detallan los patrones implementados:

3.1.5.1 Patrón Circuit Breaker

La naturaleza crítica del sistema requiere mecanismos robustos de tolerancia a fallos que garanticen la continuidad del servicio ante fallas de componentes externos. El patrón Circuit Breaker implementado proporciona degradación elegante del servicio.

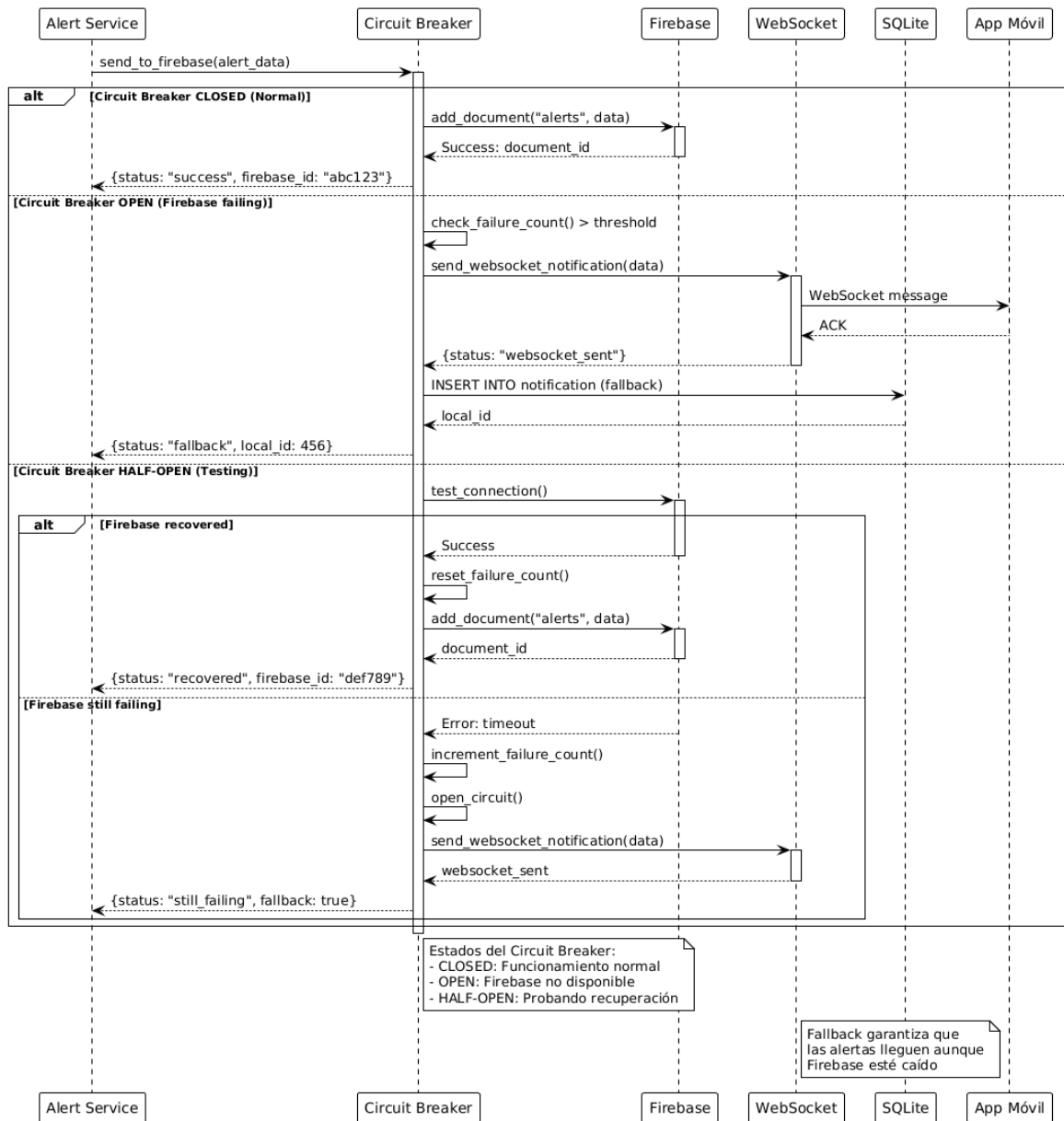


Figura 3-8 Diagrama de secuencia patrón Circuit Breaker
Fuente: Elaboración propia

El sistema implementa tres estados de operación para el Circuit Breaker:

- **Estado CLOSED:** Operación normal con Firebase disponible, sincronización completa entre SQLite y Firestore
- **Estado OPEN:** Firebase no disponible, activación de mecanismos de fallback via WebSocket para mantener comunicación con aplicaciones
- **Estado HALF-OPEN:** Período de prueba para verificar recuperación de Firebase y restauración gradual del servicio completo

3.1.5.2 Patrón Strategy

El patrón Strategy define una familia de algoritmos, los encapsula y los hace intercambiables. En el sistema, se utiliza para manejar diferentes tipos de emergencias.

Justificación técnica:

- **Extensibilidad:** Agregar nuevos tipos de emergencia (tsunami, avalancha) sin modificar código existente
- **Principio Abierto/Cerrado:** Abierto para extensión, cerrado para modificación
- **Especialización:** Cada tipo de emergencia tiene lógica y campos específicos
- **Mantenimiento:** Cambios en un tipo de emergencia no afectan otros

Beneficios obtenidos:

- Código más modular y especializado
- Facilita la adición de nuevos tipos de catástrofe
- Cada estrategia puede optimizarse independientemente

3.1.5.3 Patrón Decorator

El patrón Decorator permite agregar funcionalidad a objetos de forma dinámica sin alterar su estructura. En el sistema se utiliza para autenticación y autorización.

Justificación técnica:

- **Reutilización:** La misma lógica de autenticación se aplica a múltiples endpoints
- **Composición:** Permite combinar diferentes niveles de autorización
- **Separación de responsabilidades:** La lógica de autenticación está separada de la lógica de negocio
- **DRY (Don't Repeat Yourself):** Evita duplicar código de validación

Beneficios obtenidos:

- Código más limpio y mantenible
- Consistencia en la validación de permisos
- Facilita cambios en la lógica de autenticación

3.1.5.4 Patrón MVC (Model-View-Controller)

Django implementa oficialmente el patrón MVT, pero en el contexto de una API REST, la implementación funciona efectivamente como el patrón MVC (Model-View-Controller) clásico.

Implementación en el sistema:

- **Model (server/models.py):** Define la estructura de datos con las clases Notification y ApiKey, encapsulando toda la lógica relacionada con la persistencia y validación de datos a nivel de modelo.
- **View (api/views.py):** Contiene toda la lógica de negocio y control de flujo, manejando requests HTTP, validación de entrada, procesamiento de datos y generación de responses JSON.
- **Template → JSON Serialization:** En lugar de templates HTML tradicionales, el sistema retorna respuestas en formato JSON estructurado, manteniendo la separación de responsabilidades.

- **URL Routing (api/urls.py):** Mapea URLs específicas a vistas correspondientes, proporcionando una interfaz clara y RESTful.

Justificación técnica:

- **Separación clara de responsabilidades:** Cada capa tiene una función específica.
- **Mantenibilidad:** Cambios en una capa no afectan las otras.
- **Testabilidad:** Cada capa puede probarse independientemente.
- **Escalabilidad:** Permite optimizar cada capa según sus necesidades.

Adaptación para API REST:

- **Template → JSON Serialization:** En lugar de templates HTML, se retorna JSON.
- **View → API Endpoint:** Las vistas manejan requests HTTP y retornan responses JSON.
- **Model:** Permanece igual, representa la estructura de datos.

3.1.6 Uso de buenas prácticas

Este apartado describe los patrones de diseño aplicados en la arquitectura del sistema de alerta, con el objetivo de garantizar su escalabilidad, seguridad, mantenibilidad y facilidad de uso. A continuación, se detallan los patrones implementados:

1. Validación Exhaustiva de Entrada

El sistema implementa validación en múltiples capas: validación de autenticación mediante API keys, validación de datos obligatorios como título y descripción, y validación específica por tipo de emergencia (coordenadas obligatorias para alertas georreferenciadas).

2. Manejo Robusto de Errores

Todas las operaciones críticas están envueltas en bloques try-catch con logging detallado para debugging y graceful degradation que evita que fallos en servicios externos comprometan todo el sistema.

3. Configuración Basada en Entorno

El sistema utiliza variables de entorno para configuración, permitiendo diferentes configuraciones para desarrollo, testing y producción sin cambios en el código.

4. Principio DRY (Don't Repeat Yourself)

La lógica común se centraliza en funciones reutilizables, evitando duplicación de código y facilitando el mantenimiento.

5. Nomenclatura consistente

Se siguen convenciones de nomenclatura estándar de la guía de estilo de Python PEP 8 y Django que facilitan la comprensión del código por parte de cualquier desarrollador.

Beneficios de la Implementación

Mantenibilidad:

- Código organizado siguiendo patrones reconocidos.
- Separación clara de responsabilidades facilita localización de bugs.

Escalabilidad:

- Arquitectura modular permite crecimiento independiente de componentes.
- Patrones Strategy y Repository facilitan extensiones sin modificaciones.
- Principios SOLID garantizan flexibilidad para cambios futuros.

Testabilidad:

- Cada componente puede probarse independientemente.
- Lógica de negocio completamente separada de detalles de implementación.

Robustez:

- Manejo exhaustivo de errores previene fallos catastróficos.
- Validación en múltiples capas asegura integridad de datos.
- Logging detallado facilita debugging y monitoreo en producción.

Seguridad:

- Validación rigurosa de entrada previene inyecciones.
- Sistema de permisos granular protege operaciones sensibles.
- Trazabilidad completa de operaciones para auditoría.

3.1.7 Consideraciones de escalabilidad y mantenibilidad

Escalabilidad horizontal: El uso de Firebase permite escalar automáticamente el almacenamiento y las notificaciones con el aumento de usuarios, mientras que el backend Django puede distribuirse en múltiples servidores con un balanceador de carga.

Mantenibilidad: La modularidad del sistema, combinada con la documentación y las pruebas, asegura que futuros desarrolladores puedan entender y extender el código fácilmente.

3.1.8 Integración con el sistema general.

La integración del sistema backend de alertas de emergencia se fundamenta en una arquitectura híbrida que combina APIs REST para organismos autorizados con acceso directo a Firebase para aplicaciones móviles. Esta estrategia optimiza el rendimiento del servidor mientras garantiza la distribución eficiente de alertas críticas.

3.1.8.1 Arquitectura de Integración

El sistema implementa múltiples puntos de integración diseñados para diferentes tipos de usuarios y casos de uso.

Integración con Organismos Autorizados:

- API REST con autenticación mediante API Keys.
- Endpoints especializados para creación y consulta de alertas.
- Validación de permisos diferenciados (User/Admin).
- Respuestas estructuradas en formato JSON estándar.

Integración con Aplicación Móvil:

- Acceso directo a Firebase Firestore sin intermediación del servidor.
- Sincronización en tiempo real mediante snapshots automáticos.
- Reglas de seguridad configuradas para acceso de solo lectura.
- Optimización de ancho de banda y latencia.

Integración con Sistemas Externos:

- Webhooks para notificación a sistemas de terceros.
- Circuit Breaker para tolerancia a fallos de servicios externos.
- Fallback automático a canales alternativos de comunicación.

3.1.8.2 Especificación de API REST

El sistema expone 13 endpoints organizados en 4 módulos funcionales que cubren todas las operaciones necesarias para la gestión integral de alertas de emergencia.

El sistema de alertas de emergencia está diseñado para integrarse sin problemas con múltiples componentes externos y sistemas de gestión de emergencias, cumpliendo con los casos de uso de integración especificados en el Anexo 2. La integración se realiza mediante HTTP REST API con respuestas en formato JSON, siguiendo estándares web modernos que facilitan la interoperabilidad.

Protocolo Principal: HTTP/HTTPS REST

El sistema utiliza el protocolo HTTP/HTTPS como método principal de comunicación, implementando una arquitectura RESTful que permite la integración con cualquier sistema capaz de realizar peticiones HTTP. Esta elección se fundamenta en:

- **Universalidad:** HTTP es el protocolo estándar para comunicación web.
- **Simplicidad:** Fácil implementación en cualquier lenguaje de programación.
- **Escalabilidad:** Permite balanceadores de carga y proxies reversos.
- **Seguridad:** Soporte nativo para HTTPS y autenticación mediante headers.

Formato de Intercambio: JSON

Todas las comunicaciones utilizan JSON como formato de intercambio de datos, proporcionando:

- **Legibilidad:** Formato human-readable que facilita debugging.
- **Compatibilidad:** Soporte nativo en prácticamente todos los lenguajes modernos.
- **Eficiencia:** Menor overhead comparado con XML.
- **Validación:** Esquemas JSON para validación de estructura.

Endpoint	Método	Permisos	Caso de Uso	Descripción
/api/send-notification/	POST	User	CU-01	Envío de alertas de emergencia
/api/alerts/	GET	User	CU-02	Listado de alertas existentes
/api/alerts/{id}/	GET	User	CU-02	Consulta de alerta específica
/api/alerts/{id}/update/	PUT	Admin	CU-03	Actualización de alerta existente
/api/alerts/{id}/delete/	DELETE	Admin	CU-03	Eliminación de alerta
/auth/validate/	POST	Público	CU-10	Validar API key y verificar permisos
/api/api-keys/	GET	Admin	CU-06	Listado de API keys
/api/api-keys/create/	POST	Admin	CU-06	Creación de nueva API key
/api/api-keys/{id}/revoke/	PUT	Admin	CU-06	Revocación de API key
/geo/validate-coordinates/	POST	User	CU-05	Validar coordenadas geográficas de entrada
/geo/calculate-evacuation-zone/	POST	User	CU-05	Calcular zona completa de evacuación
/geo/calculate-evacuation-radius/	POST	User	CU-05	Calcular radio según tipo de emergencia
/geo/find-evacuation-points/	POST	User	CU-05	Encontrar puntos seguros de evacuación

Tabla 3-3 Endpoints API REST
Fuente: Elaboración propia

Parámetros de Endpoints

Parámetro	Tipo	Requerido	Descripción
title	string	si	Título de la alerta (max 255 chars)
body	string	si	Descripción detallada de la emergencia
type	string	si	Tipo: "incendio", "earthquake", "general"
latitude	float	no	Latitud del evento (-90 a 90)
longitude	float	no	Longitud del evento (-180 a 180)
magnitude	float	no	Magnitud (requerido si type="earthquake")
Active	boolean	no	Estado activo (default: true)

Tabla 3-4 POST /api/send-notification/ Body Parameters (JSON)
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
active	boolean	no	Filtrar por estado activo/inactivo
type	string	no	Filtrar por tipo de emergencia
limit	Int	no	Número de resultados (default: 20, max: 100)
offset	Int	no	Desplazamiento para paginación (default: 0)

Tabla 3-5 GET /api/alerts/ Query Parameters
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
Id	int	si	ID único de la alerta

Tabla 3-6 GET /api/alerts/{id}/ Path Parameters
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
Id	int	si	ID único de la alerta

Tabla 3-7 PUT /api/alerts/{id}/update/ Path Parameters
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
title	string	no	Nuevo título de la alerta
body	string	no	Nueva descripción
active	boolean	no	Cambiar estado activo/inactivo
latitude	float	no	Nueva latitud
longitude	float	no	Nueva longitud

Tabla 3-8 PUT /api/alerts/{id}/update/ Body Parameters (JSON)
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
id	int	si	ID único de la alerta a eliminar

Tabla 3-9 DELETE /api/alerts/{id}/delete/ Path Parameters
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
api_key	string	si	UUID de la API key a validar

Tabla 3-10 POST /auth/validate/ Body Parameters (JSON)
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
active	boolean	no	Filtrar por estado activo/inactivo
is_admin	boolean	no	Filtrar por nivel de permisos
limit	int	no	Número de resultados (default: 20)
offset	int	no	Desplazamiento para paginación

Tabla 3-11 GET /api/api-keys/ Query Parameters
Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
name	string	si	Nombre identificador técnico
entidad	string	si	Nombre oficial del organismo
description	string	no	Descripción opcional del uso
is_admin	boolean	no	Permisos de administrador (default: false)

Tabla 3-12 POST /api/api-keys/create/ Body Parameters (JSON)

Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
id	int	si	ID único de la API key

Tabla 3-13 PUT /api/api-keys/{id}/revoke/ Path Parameters

Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
latitude	float	si	Latitud a validar (-90 a 90)
longitude	float	si	Longitud a validar (-180 a 180)

Tabla 3-14 POST /geo/validate-coordinates/ Body Parameters (JSON)

Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
latitude	float	si	Latitud del centro de la emergencia
longitude	float	si	Longitud del centro de la emergencia
type	string	si	Tipo: "incendio" o "earthquake"

Tabla 3-15 POST /geo/calculate-evacuation-zone/ Body Parameters (JSON)

Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
type	string	si	Tipo de emergencia: "incendio" o "earthquake"

Tabla 3-16 POST /geo/calculate-evacuation-radius/ Body Parameters (JSON)

Fuente: Elaboración propia

Parámetro	Tipo	Requerido	Descripción
latitude	float	si	Latitud del punto de referencia
longitude	float	si	Longitud del punto de referencia
radius_km	float	no	Radio de búsqueda en km (default: 2)

Tabla 3-17 POST /geo/find-evacuation-points/ Body Parameters (JSON)

Fuente: Elaboración propia

3.2 Detalles de la codificación y desarrollo.

El desarrollo del sistema de alertas de emergencia se organizó mediante metodología Scrum con 3 sprints de 1 mes cada uno. Sin embargo, el equipo enfrentó dificultades técnicas y de coordinación que limitaron el alcance del sistema presentado en la feria de software, priorizando las funcionalidades más esenciales para demostrar la viabilidad del concepto.

3.2.1 Desafíos Encontrados Durante el Desarrollo

Dificultades Técnicas:

- **Complejidad de integración con Firebase en entornos de desarrollo diversos:** Cambios iniciales en la exploración de tecnologías llevaron a una pérdida de tiempo de desarrollo significativo en el primer sprint, llegando al stack final. También el cambio de sistema operativo de Windows a Linux en el tiempo de desarrollo posterior a la feria de software requirió cambios menores pero necesarios para mantener el funcionamiento del sistema.
- **Configuración de WebSocket para comunicación en tiempo real:** La implementación inicial de este protocolo demostró ser más compleja de lo anticipado, específicamente en lo relacionado con la gestión de conexiones persistentes.
- **Sincronización entre almacenamiento local y en la nube:** Mantener la consistencia de datos entre la base de datos SQLite local y Firebase Firestore requirió desarrollar lógica adicional de sincronización que no estaba contemplada inicialmente en el diseño.
- **Implementación de patrones de tolerancia a fallos:** La incorporación de mecanismos como Circuit Breaker para manejar fallos en servicios externos añadió complejidad al código y requirió varias iteraciones para lograr un comportamiento estable.

Dificultades de Coordinación:

- **Coordinación de trabajo remoto entre miembros del equipo:** La falta de interacción por parte de miembros del equipo y trabajo remoto dificultó la resolución rápida de dudas técnicas y la alineación en decisiones de implementación, requiriendo una comunicación más estructurada y frecuente.
- **Diferencias en configuración de entornos de desarrollo:** La falta de una configuración estandarizada inicial generó inconsistencias donde el código funcionaba en un equipo, pero requería de asistencia para funcionar en otro.
- **Tiempo limitado para implementación completa de microservicios:** La ambiciosa arquitectura planteada inicialmente resultó imposible de completar dentro de los plazos, forzando al equipo a postergar la migración a microservicios y mantener una arquitectura monolítica para la presentación en la feria.

3.2.2 Estado del Sistema en la Feria de Software

Funcionalidades Logradas (MVP Presentado)

- Autenticación básica: Sistema de API keys funcional
- Generación de API keys
- Envío de alertas
- Listado de alertas
- Almacenamiento dual: SQLite local + Firebase Firestore
- Tipos de emergencia: Soporte para incendios y terremotos
- Validación de datos: Campos obligatorios y coordenadas geográficas
- Integración Firebase: FCM para notificaciones push básicas

Funcionalidades No Logradas (Pendientes Post-Feria)

- Separación total de servicios
- Tolerancia a fallos automática
- Comunicación bidireccional
- Cálculos de evacuación automáticos
- Actualización de alertas por API
- Eliminación de alertas por API
- Testing automatizado
- Automatización de despliegue

3.2.3 Estrategia de Presentación en la Feria

El equipo logró llevar a la feria de software un producto mínimo viable que demostraba el funcionamiento de la aplicación.

Valor Demostrado:

- **Viabilidad técnica:** Integración exitosa Django + Firebase
- **Funcionalidad crítica:** Envío y almacenamiento de alertas
- **Escalabilidad base:** Arquitectura preparada para crecimiento
- **Seguridad básica:** Control de acceso mediante API keys

3.2.4 Desarrollo Post-Feria

Funcionalidades Implementadas Posteriormente:

- Circuit Breaker Pattern para tolerancia a fallos
- WebSocket para comunicación en tiempo real
- Servicios geoespaciales con cálculos de evacuación
- Suite de pruebas unitarias
- Scripts de automatización

Funcionalidades pendientes:

- Arquitectura de microservicios completa

3.2.5 Lecciones Aprendidas

Gestión de Proyecto:

- **Priorización efectiva de funcionalidades core para demostración:** Se enfocaron los esfuerzos en desarrollar primero las características indispensables lo que permitió presentar un producto viable en la feria.
- **Importancia de MVP funcional sobre funcionalidades avanzadas:** Comprobar que es mejor tener un producto básico pero operativo, que uno con muchas características. Esto permitió validar el concepto con usuarios reales y obtener feedback temprano.
- **Valor de los scripts de demostración automatizados:** La preparación de datos de prueba preconfigurados fue necesario para realizar demostraciones sin contratiempos durante la feria, mostrando el sistema en su mejor luz.

Desarrollo Técnico:

- **Configuración de entornos de desarrollo estandarizada:** Estandarizar los entornos con entornos virtuales y requisitos, ayudo a portar el sistema de una maquina a otra sin mayores problemas.
- **Integración temprana con servicios externos críticos:** Conectar el sistema con Firebase desde el principio permitió identificar y resolver problemas de integración en fases iniciales, evitando comportamientos inesperados en etapas avanzadas del desarrollo.
- **Testing continuo de funcionalidades esenciales:** Implementar pruebas automatizadas para las funciones críticas garantizó que el sistema se mantenga operativo al añadir nuevas funcionalidades.

Metodología Scrum Adaptada:

- **Sprint 1:** Infraestructura y funcionalidad básica (logrado)
- **Sprint 2:** Refinamiento para feria (logrado parcialmente)
- **Sprint 3:** Funcionalidades avanzadas (completado post-feria)

Esta experiencia demostró la importancia de entregar un MVP funcional que valide el concepto técnico, obtener feedback, así permitiendo desarrollo iterativo posterior para completar funcionalidades avanzadas una vez validada la viabilidad del sistema.

3.3 Pruebas y Validación

La estrategia de pruebas del sistema de alertas de emergencia se diseñó considerando la naturaleza crítica del dominio, donde fallos pueden tener consecuencias graves. Se implementó un enfoque de testing multinivel que abarca desde pruebas unitarias hasta validación de integración con servicios externos.

3.3.1 Estrategia de Testing

La estrategia de testing del sistema de alertas de emergencia se diseñó considerando la naturaleza crítica del dominio, donde fallos pueden tener consecuencias graves. Se adoptó un enfoque multinivel que garantiza la confiabilidad de cada componente y del sistema integrado.

Pirámide de Pruebas Implementada:

- **Pruebas Unitarias (70%):** Validación de componentes individuales y lógica de negocio
- **Pruebas de Integración (20%):** Comunicación entre servicios internos y servicios externos
- **Pruebas de Sistema (10%):** Flujos completos end-to-end y casos de uso críticos

Herramientas Utilizadas:

- **Django TestCase:** Framework principal para pruebas unitarias
- **Scripts automatizados:** Validación de casos de uso reales
- **Firebase Console:** Verificación de integración en tiempo real
- **Postman:** Testing manual de APIs REST

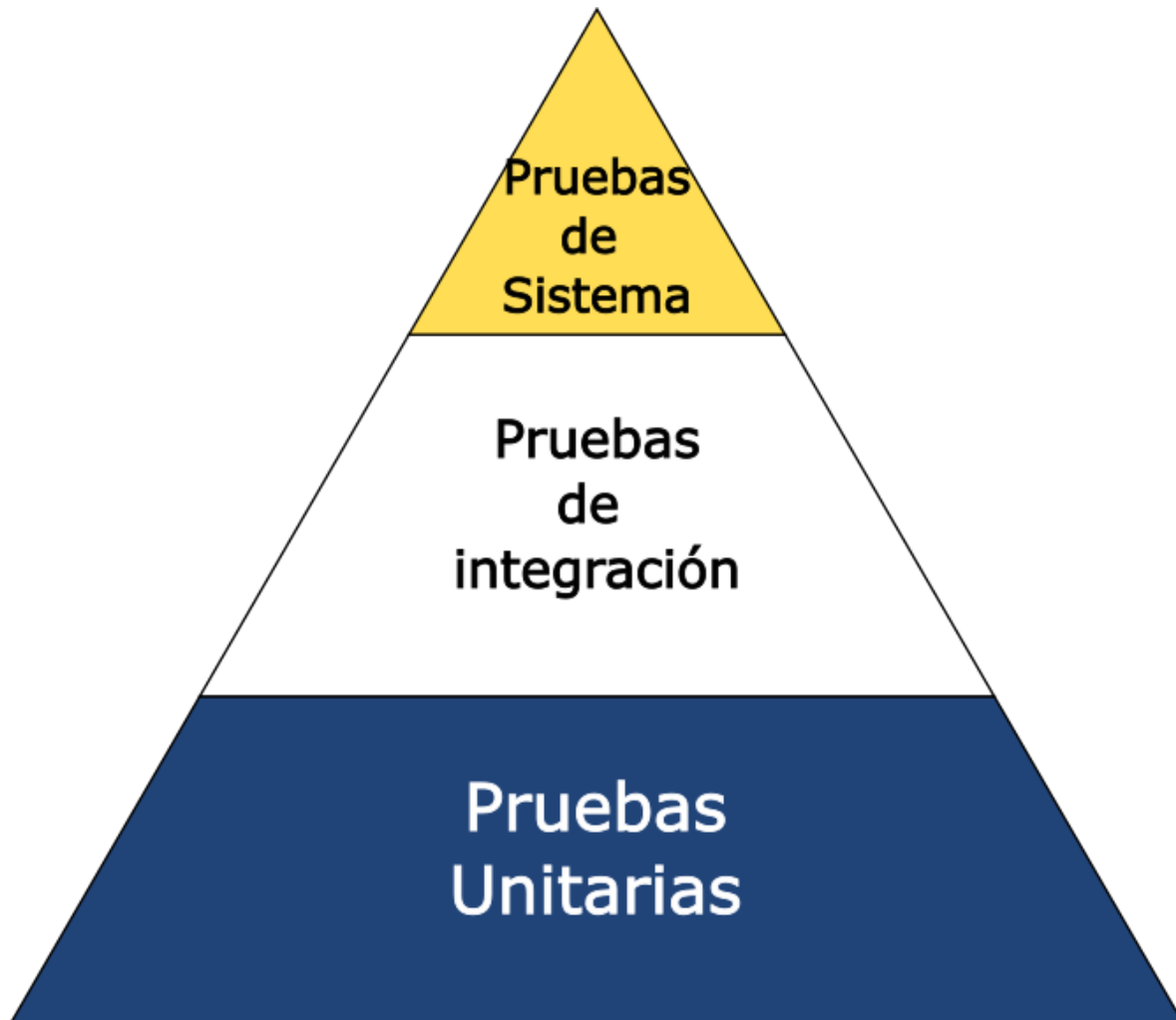


Figura 3-9 Pirámide de pruebas
Fuente: Elaboración Propia

3.3.2 Estrategias de testing aplicadas a los componentes.

Para garantizar la robustez y confiabilidad de cada parte del sistema, se diseñó y ejecutó una estrategia de pruebas específica para sus componentes principales. A continuación, se detallan las estrategias y técnicas de testing aplicadas a cada uno de ellos.

3.3.2.1 API Gateway y Sistema de Autenticación

Las pruebas unitarias implementadas con Django TestCase se enfocan en la validación de tokens API key con diferentes formatos, verificando tanto UUID válidos como inválidos. Se incluye la verificación de headers HTTP requeridos como Content-Type y X-API-Key, junto con el manejo de excepciones para claves expiradas o desactivadas y la sanitización de entrada de datos JSON malformados.

La validación con Postman complementa estas pruebas mediante:

- Casos exitosos de autenticación con API keys válidas
- Verificación de errores 401 para claves inválidas o ausentes
- Validación del formato de respuestas JSON y testing de diferentes métodos HTTP

Todas las solicitudes de Postman se encuentran disponibles en el Anexo 3.

Esta combinación asegura que el sistema de autenticación funcione correctamente tanto a nivel de componente como de integración.

3.3.2.2 Procesador de Alertas de Emergencia

Las pruebas unitarias del procesador se centran en la validación de tipos de catástrofe soportados, verificación de campos obligatorios específicos para cada tipo de alerta, transformación correcta de datos de entrada al formato requerido por Firestore, y el cálculo automático de radios de evacuación según el tipo de emergencia.

El testing con Postman valida el comportamiento end-to-end mediante el envío de alertas de incendio con coordenadas válidas, creación de alertas de terremoto con magnitud específica, y validación de campos requeridos como title, body y type. Se incluyen casos límite con coordenadas extremas y magnitudes altas para garantizar robustez del sistema.

3.3.2.3 Servicio de Almacenamiento

Las pruebas unitarias del servicio de almacenamiento verifican la serialización correcta de documentos de alerta para Firestore, validación de esquemas de datos específicos por tipo de catástrofe, manejo apropiado de timestamps y formato de coordenadas, junto con la verificación de estructura correcta de arrays como puntos_evacuacion y radios_evacuacion.

La verificación en Firebase Console permite:

- Inspección manual de documentos creados en la colección "alertas"
- Validación visual de la estructura de datos para incendios y terremotos
- Verificación de timestamps automáticos generados por SERVER_TIMESTAMP
- Confirmación de tipos de datos correctos

Esta verificación directa en la consola de Firebase proporciona confianza en la integridad de los datos almacenados.

3.3.2.4 Sistema de Notificaciones

Las pruebas unitarias del sistema de notificaciones abarcan el formateo correcto de mensajes para diferentes canales, validación de topics y estructura de payloads, manejo apropiado de errores de Firebase Cloud Messaging, y verificación de la persistencia en la base de datos local SQLite.

El testing integral con Postman confirma las respuestas exitosas con status 200, verificación de flags de estado como firestore_saved: true, validación de metadatos de respuesta incluyendo source y alert_type, además del testing secuencial de diferentes tipos de alerta para verificar la consistencia del comportamiento.

3.3.2.5 Casos de Prueba y Herramientas de Validación

Los escenarios funcionales implementados incluyeron alertas de incendio para USM Sede Viña del Mar con coordenadas específicas, alertas de terremoto, y validación de errores mediante campos faltantes, API keys inválidas y JSON malformado.

La verificación de persistencia se realizó mediante inspección directa en Firebase Console de documentos en Firestore, verificación de registros en SQLite a través del Django admin, y comparación de consistencia entre almacenamiento local y remoto.

Las herramientas principales utilizadas fueron:

- Django TestCase para pruebas unitarias de componentes individuales
- Postman para testing manual de endpoints REST y validación de respuestas
- Firebase Console para verificación visual de datos almacenados

3.3.2.6 Criterios de Aceptación y Validación de Integridad

La funcionalidad crítica del sistema requiere que todas las alertas persistan correctamente en Firestore, que la API responda en menos de 2 segundos, que la validación de autenticación sea 100% efectiva, y que la estructura de datos cumpla las especificaciones exactas definidas.

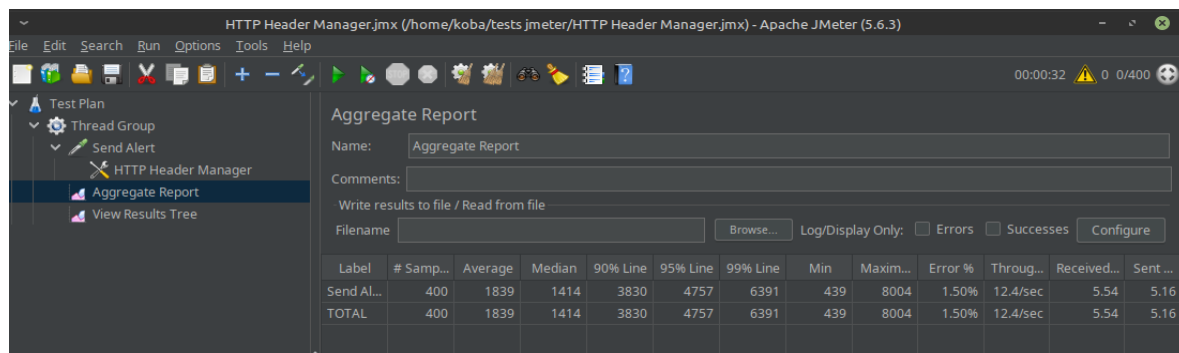
La validación de integridad asegura que los documentos en Firestore coincidan con la especificación técnica, que los campos obligatorios se validen correctamente, que los tipos de datos sean consistentes, y que los timestamps se generen automáticamente. Esta estrategia de testing garantiza la funcionalidad core del sistema mediante validación directa de componentes críticos y verificación manual de resultados en las herramientas de producción.

3.3.2.7 Pruebas de estrés

Para validar la capacidad del sistema de manejar múltiples alertas simultáneas, se realizaron pruebas de estrés utilizando Apache JMeter. Las pruebas simularon 400 usuarios concurrentes enviando alertas de incendio y terremoto durante un período de ramp-up de 30 segundos, evaluando el tiempo de respuesta, throughput y tasa de errores del sistema.

Los resultados iniciales con el servidor de desarrollo Django mostraron una tasa de errores de 1.5% al ejecutar las pruebas. Estos valores son aceptables en este ambiente de prueba, pero deben ser observados al pasar el sistema a producción.

La captura de pantalla siguiente muestra el reporte posterior a la prueba de estrés ejecutada sobre el servidor.



Label	# Samp...	Average	Median	90% Line	95% Line	99% Line	Min	Maxim...	Error %	Throug...	Received...	Sent ...
Send Al...	400	1839	1414	3830	4757	6391	439	8004	1.50%	12.4/sec	5.54	5.16
TOTAL	400	1839	1414	3830	4757	6391	439	8004	1.50%	12.4/sec	5.54	5.16

Figura 3-10 Prueba de estrés JMeter

Fuente: Elaboración Propia

3.3.3 Resolución de bugs y optimización

El proceso de desarrollo incluyó un ciclo continuo de identificación, diagnóstico y resolución de bugs, así como la optimización del código para mejorar el rendimiento y la confiabilidad. Entre los problemas más significativos que se abordaron se encontraron:

- **Control de Accesos para la Creación de API Keys:** Se descubrió un error de seguridad en el endpoint de creación de claves API, donde usuarios sin permisos de administrador podían generar nuevas credenciales. Esto se solucionó reforzando el middleware de autenticación y autorización, implementando una verificación estricta del rol de administrador (`is_admin`) antes de permitir cualquier operación de escritura en el modelo `ApiKey`.
- **Validación de Coordenadas Geográficas:** Inicialmente, el sistema permitía el guardado de coordenadas geográficas inválidas (fuera de los rangos -90 a 90 para latitud y -180 a 180 para longitud). Se implementó un validador específico en el modelo y a nivel del serializador de la API que rechaza cualquier coordenada fuera de estos rangos, garantizando la integridad de los datos geoespaciales.

Se consideró la implementación de caché para consultas frecuentes, pero no fue implementado debido a restricciones de temporales.

3.4 Integración con Otros Componentes

Esta sección describe las estrategias y mecanismos implementados para garantizar una comunicación fluida y segura entre los diferentes módulos del sistema y con plataformas externas de gestión de emergencias.

3.4.1 Arquitectura de Integración

El sistema de alertas de emergencia está diseñado como una plataforma centralizada que facilita la integración con organismos oficiales de emergencia mediante APIs REST estándar. La arquitectura permite que múltiples entidades autorizadas envíen alertas de manera coordinada y segura.

Modelo de Integración:

- **API Gateway Centralizado:** Punto único de entrada para todos los organismos
- **Autenticación por API Keys:** Control granular de acceso por organización
- **Protocolo HTTP/REST:** Estándar ampliamente soportado por sistemas gubernamentales
- **Formato JSON:** Intercambio de datos estructurado y legible

4 Conclusiones

El desarrollo del sistema backend Alertify demostró la viabilidad de crear una plataforma escalable y robusta para la gestión de alertas de emergencia, superando limitaciones de sistemas tradicionales como el SAE. Se implementó con éxito una API REST segura que permite el envío de alertas detalladas y georreferenciadas, junto con un sistema de almacenamiento híbrido que combina SQLite para consistencia transaccional y Firebase Firestore para distribución en tiempo real. Se integraron componentes clave como geolocalización, notificaciones push y autenticación mediante API keys, cumpliendo con los objetivos de entregar información contextualizada y accionable durante emergencias.

El proyecto enfrentó desafíos técnicos y de coordinación, lo que llevó a priorizar un producto mínimo viable funcional para su presentación en la feria de software. Si bien no se implementaron todas las

funcionalidades inicialmente planificadas, como microservicios independientes o rutas de evacuación automáticas, el sistema demostró capacidad de integración, tolerancia a fallos básica y una arquitectura preparada para futuras extensiones. Esta experiencia reforzó la importancia de desarrollar un núcleo operativo confiable antes de incorporar funcionalidades avanzadas.

Como trabajo futuro, se recomienda migrar a una arquitectura de microservicios para mejorar la escalabilidad, integrar fuentes de datos en tiempo real como sensores IoT, y agregar capacidades predictivas mediante machine learning. Alertify sienta las bases para un sistema de alertas inteligente que puede reducir tiempos de respuesta, optimizar evacuaciones y salvar vidas mediante tecnología accesible y confiable.

5 Agradecimientos

En primer lugar, agradezco a mi Profesor Guía, Carlos Alten, por su orientación, apoyo y paciencia a lo largo del desarrollo de esta tesina. Su retroalimentación fue fundamental para dar forma y dirección a este trabajo.

A mis compañeros de equipo, Camilo Leal y Paulina Candia, por su apoyo y compromiso durante las etapas más desafiantes del desarrollo del proyecto.

A mis amigos, a la Taberna y todos quienes estuvieron cuando más lo necesité.

Finalmente, a mi familia por el apoyo emocional y moral a lo largo de este proceso, en especial a mi Aguela Mona que hubiese estado muy feliz con que terminase esta etapa de mi vida.

6 Referencias

- [1] «Informe | Incendios 02 y 03 de febrero de 2024 Viña del Mar (Región Valparaíso),» [En línea]. Available: <https://wildfirex.cl/informe-incendios-02-y-03-de-febrero-de-2024-vina-del-mar-region-valparaiso/>.
- [2] «Qué es el SAE: así funciona el Sistema de Alerta de Emergencias de Senapred,» [En línea]. Available: <https://www.latercera.com/servicios/noticia/que-es-el-sae-asi-funciona-el-sistema-de-alerta-de-emergencias-de-senapred/SRSPJGP47BDONAU5CFCOKJN5E4/>.
- [3] M. D. L'Herbe, «Ciper,» 10 Marzo 2024. [En línea]. Available: <https://www.ciperchile.cl/2024/03/10/incendios-el-fracaso-del-sae/>.
- [4] «Las diferencias entre Scrum y otros métodos ágiles,» [En línea]. Available: <https://www.gestionproyectos.es/las-diferencias-entre-scrum-y-otros-metodos-agiles/>.
- [5] Amazon, «Front End frente a back-end: diferencia entre el desarrollo de aplicaciones - AWS,» [En línea]. Available: <https://aws.amazon.com/es/compare/the-difference-between-frontend-and-backend/>.
- [6] «¿Qué son los microservicios? | AWS,» [En línea]. Available: <https://aws.amazon.com/es/microservices/>.
- [7] «¿Qué es la computación en la nube? - Servicios, ventajas y tipos de computación en la nube - AWS,» [En línea]. Available: <https://aws.amazon.com/es/what-is-cloud-computing/>.

- [8] «¿Qué es una base de datos NoSQL? - Explicación de las bases de datos no relacionales - AWS,» [En línea]. Available: <https://aws.amazon.com/es/nosql/>.
- [9] «¿Qué es una API? Explicación de qué es una interfaz de programación de aplicaciones. AWS,» [En línea]. Available: <https://aws.amazon.com/es/what-is/api/>.
- [10] «¿Cómo funciona el modelo cliente-servidor?,» [En línea]. Available: <https://www.ionos.mx/digitalguide/servidores/know-how/modelo-cliente-servidor/>.
- [11] «Patrones de arquitectura de software en el diseño de sistemas,» [En línea]. Available: <https://www.geeksforgeeks.org/system-design/design-patterns-architecture/>.
- [12] «Microservice architecture vs modular architecture,» [En línea]. Available: <https://www.developer-tech.com/news/microservice-architecture-vs-modular-architecture/>.
- [13] «¿Qué es una API gateway? | IBM,» [En línea]. Available: <https://www.ibm.com/mx-es/think/topics/api-gateway>.
- [14] «Circuit breaker pattern - AWS Prescriptive Guidance,» [En línea]. Available: <https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/circuit-breaker.html>.
- [15] «¿Qué es la arquitectura basada en eventos? | IBM,» [En línea]. Available: <https://www.ibm.com/mx-es/think/topics/event-driven-architecture>.
- [16] «Comparativa de Frameworks Backend 2025: Arquitectando Aplicaciones a Prueba de Futuro,» [En línea]. Available: <https://blog.madrigan.com/es/blog/202510031846/>.
- [17] «Firebase | Google's Mobile and Web App Development Platform,» [En línea]. Available: <https://firebase.google.com/>.
- [18] «Cómo elegir tu base de datos: Cloud Firestore o Realtime Database,» [En línea]. Available: <https://firebase.google.com/docs/database/rtdb-vs-firestore?hl=es-419>.
- [19] «Firebase Cloud Messaging,» [En línea]. Available: <https://firebase.google.com/docs/cloud-messaging?hl=es-419>.
- [20] «Why Use Python? - Full Stack Python,» [En línea]. Available: <https://www.fullstackpython.com/why-use-python.html>.
- [21] «Django Channels — Channels 4.3.1 documentation,» [En línea]. Available: <https://channels.readthedocs.io/en/latest/>.
- [22] «Scrum Guides,» [En línea]. Available: <https://scrumguides.org/>.
- [23] «Arquitectura de microservicios | Atlassian,» [En línea]. Available: <https://www.atlassian.com/es/microservices/microservices-architecture>.

7 Anexos

7.1 Anexo 1: Presentación en Feria de Software USM

Este anexo contiene la cobertura oficial de la Universidad Técnica Federico Santa María sobre la presentación del proyecto "Sistema de Alertas de Catástrofe" en la Feria de Software 2024. El artículo documenta la participación del equipo de desarrollo y presenta el sistema como una solución innovadora para la gestión de emergencias en tiempo real.

<https://usm.cl/noticias/estudiantes-crean-app-para-obtener-informacion-en-tiempo-real-durante-catastrofes/>



7.2 Anexo 2: Casos de uso

Especificación completa de los 10 casos de uso del sistema, incluyendo flujos principales, precondiciones y postcondiciones para cada funcionalidad implementada.

https://drive.google.com/drive/folders/18EwjutpWsB_gwePieavhnwsQWwMv0MvT

7.3 Anexo 3: Colección de Pruebas API (Postman)

Este anexo contiene la colección completa de requests de Postman utilizada para el testing y validación de la API REST del sistema. La colección incluye todos los endpoints implementados con ejemplos de uso prácticos y casos de prueba reales.

Contenido de la colección:

- **13 endpoints documentados** con requests de ejemplo.
- **Headers de autenticación** configurados (X-API-Key).
- **Casos de prueba positivos y negativos** para cada endpoint.
- **Variables de entorno** para facilitar testing en diferentes ambientes.
- **Ejemplos de payloads JSON** con datos reales de emergencias.
- **Validaciones de respuesta** automatizadas.

https://raw.githubusercontent.com/itsBocchi/itsBocchi/refs/heads/main/Sistema%20de%20Alertas%20de%20Cat%C3%A1strofe.postman_collection.json