### Documento de Arquitectura del Sistema de Seguridad Avanzado para Urbanización

**Departamento Nacional de Planeación** **Bogotá, 2025**

### HISTORIAL DE VERSIONES

| Versión | Fecha | Descripción | Autor |
| --- | --- | --- | --- |
| 1.0 | 2025-09-16 | Versión Inicial del Documento de Arquitectura | [Andres Felipe Chicaiza Palechor,Santiago Lopez León, Eduar Ruiz ] |

Exportar a Hojas de cálculo

### CUADRO DE ELABORACIÓN Y APROBACIÓN

| Elaborado Por | Revisado Por | Aprobado Por |
| --- | --- | --- |
| [Tu Nombre] | [Nombre del Arquitecto / Líder] | [Nombre del Gerente de Proyecto] |
| Cargo: Desarrollador Principal | Cargo: Arquitecto de Software | Cargo: Gerente de Proyecto |

Exportar a Hojas de cálculo

### 1. INTRODUCCIÓN

El presente documento describe la arquitectura del sistema de seguridad avanzado para un conjunto residencial, un proyecto que busca optimizar la gestión de la seguridad a través de tecnología. El sistema se centra en el monitoreo de cámaras (CCTV), el control de acceso vehicular por reconocimiento de placas, la gestión de rondas de celadores y la comunicación en tiempo real con los residentes. La arquitectura está diseñada para ser modular, escalable y mantenible, utilizando principios de diseño sólidos y patrones de arquitectura.

### 2. OBJETIVO

El objetivo principal de este documento es plasmar las decisiones de diseño arquitectónico que guían el desarrollo e implementación del sistema. Esto incluye definir los componentes, las relaciones entre ellos, los patrones de diseño aplicados y las vistas de despliegue, asegurando que la solución final cumpla con los requerimientos funcionales y no funcionales de seguridad y eficiencia.

### 3. ALCANCE

El sistema abarca las siguientes funcionalidades principales, distribuidas en diferentes interfaces de usuario:

* **Control de Accesos Vehiculares:** Verificación y registro de vehículos a través de una API de reconocimiento de placas.
* **Monitoreo de CCTV:** Visualización de cámaras de seguridad por parte del operador.
* **Gestión de Rondas:** Registro de rondas de seguridad por el celador.
* **Gestión de Registros:** Un panel de administración para visualizar y gestionar el historial de accesos.
* **Comunicación con Residentes:** Un portal para que los residentes consulten información y se comuniquen con el personal de seguridad.
* **Notificaciones en Tiempo Real:** Sistema de notificaciones para comunicar eventos clave a los usuarios.
* **Cálculo de Rutas Internas:** Un algoritmo de búsqueda bidireccional para optimizar las rondas o guiar a los visitantes.

### 4. TÉRMINOS Y DEFINICIONES

* **API (Application Programming Interface):** Conjunto de reglas que permiten a una aplicación comunicarse con otra. En este sistema, la API es el backend desarrollado en Flask (app.py).
* **Búsqueda Bidireccional:** Un algoritmo de búsqueda de ruta que opera simultáneamente desde el nodo inicial y el nodo objetivo para encontrar el camino más corto de manera eficiente.
* **EventBus:** Un patrón de comunicación (implementación del Patrón Observer) que permite a diferentes componentes interactuar entre sí sin estar directamente acoplados, utilizando eventos y suscriptores.
* **Flask:** Un micro-framework web de Python utilizado para construir la capa de lógica de negocio (backend).
* **Principios SOLID:** Conjunto de cinco principios de diseño orientados a objetos que promueven un código más comprensible, flexible y mantenible.

### 5. DOCUMENTO DE ARQUITECTURA

#### 5.1 Vista Lógica (Patrones de Diseño y Principios SOLID)

El sistema se estructura siguiendo los principios SOLID para garantizar un código modular y de fácil mantenimiento.

* **Principio de Responsabilidad Única (SRP):** Cada componente tiene una única responsabilidad. Por ejemplo, app.py se encarga de la comunicación con la base de datos y la gestión de los endpoints de la API, mientras que python.py maneja la lógica de negocio como los algoritmos de búsqueda y la comunicación de eventos.
* **Patrón Observer (EventBus):** El sistema implementa este patrón a través de un EventBus (python.py). Los componentes (observadores) se suscriben a eventos específicos (acceso, ronda, alerta). Cuando un evento es publicado, el EventBus notifica automáticamente a todos los suscriptores. Por ejemplo, al registrar un acceso en Celador.html, se publica un evento que podría ser capturado por un sistema de registro de logs o un notificador.

#### 5.2 Vista de Despliegue

La arquitectura de despliegue se basa en un modelo monolítico de tres capas:

1. **Capa de Presentación (Frontend):** Constituida por los archivos HTML (index.html, Administrador.html, Celador.html, OperadorSeguridad.html, Residentes.html) que representan las interfaces de usuario. Estas páginas web se ejecutan en los navegadores de los clientes.
2. **Capa de Lógica de Negocio (Backend):** Servidor Flask (app.py) que expone una API REST para gestionar la lógica de negocio y la comunicación con la base de datos.
3. **Capa de Datos:** Una base de datos **PostgreSQL** que almacena los registros de accesos, información de residentes y otros datos críticos.

El despliegue inicial es local, con el servidor Flask ejecutándose en http://localhost:5000 y la base de datos en el mismo host.

#### 5.3 Vista de Componentes

| Componente | Rol | Tecnología | Descripción |
| --- | --- | --- | --- |
| **app.py** | **Backend API** | Python (Flask) | Maneja las peticiones HTTP (GET, POST), se conecta a la base de datos (PostgreSQL) y ejecuta las consultas. Es el motor principal del sistema. |
| **python.py** | **Lógica de Negocio** | Python | Contiene el EventBus para la comunicación entre componentes y el algoritmo de búsqueda bidireccional para el cálculo de rutas. Es el cerebro de las operaciones complejas. |
| **Administrador.html** | **Frontend** | HTML, CSS, JS | Interfaz de gestión para el administrador, permite visualizar el historial de accesos y otros registros. |
| **Celador.html** | **Frontend** | HTML, CSS, JS | Interfaz para el personal de vigilancia, permite verificar placas de vehículos y registrar rondas. |
| **OperadorSeguridad.html** | **Frontend** | HTML, CSS, JS | Interfaz para el operador de seguridad, con funciones de monitoreo de cámaras y generación de alertas. |
| **Residentes.html** | **Frontend** | HTML, CSS, JS | Portal para residentes que permite ver transmisiones de cámaras y enviar mensajes. |
| **Base de Datos** | **Capa de Datos** | PostgreSQL | Almacena información de accesos, residentes y vehículos autorizados. |

Exportar a Hojas de cálculo

#### 5.4 Vista de Procesos

El flujo de verificación de acceso es un ejemplo del proceso dentro del sistema:

1. El celador ingresa una placa en el campo de texto de Celador.html y presiona "Verificar".
2. El JavaScript en Celador.html realiza una llamada fetch (GET) al endpoint http://localhost:5000/api/verificar-acceso/<placa>.
3. El servidor Flask, en app.py, recibe la petición y se conecta a la base de datos.
4. Ejecuta una consulta SQL para validar la placa en la tabla de residentes.
5. Basado en el resultado de la consulta, app.py devuelve una respuesta JSON indicando si el acceso es autorizado o denegado.
6. El JavaScript en Celador.html procesa la respuesta y actualiza la interfaz de usuario para mostrar el resultado.

6. CONCLUSIÓN

La arquitectura del sistema de seguridad, sustentada en los principios SOLID y patrones de diseño como el Observer, asegura un desarrollo robusto, escalable y fácil de mantener. La clara separación de responsabilidades entre el frontend, el backend y la base de datos permite la futura evolución del sistema y su adaptación a nuevas tecnologías o requerimientos. El uso de algoritmos avanzados, como la búsqueda bidireccional, demuestra un enfoque integral en la optimización de los procesos de seguridad.