Informe de Documentación del Proyecto: Sistema de Seguridad Concurrente (Stark Industries)

Programación Concurrente Guillermo García Peryona

Índice de Contenidos

1. Contexto
2. Objetivo del Proyecto
3. Diagnóstico Ejecutivo
4. Análisis Técnico Detallado
   * main.py: Orquestador de la Aplicación y API
   * security.py: Módulo de Autenticación y Autorización
   * sensors.py: Modelos de Dominio y Lógica de Sensores
   * processing.py: Gestión de Concurrencia y Lógica de Negocio
   * websocket.py: Gestor de Conexiones en Tiempo Real
   * metrics.py: Monitorización y Observabilidad
   * static/: Interfaz de Usuario (Frontend)
5. Mejoras de Calidad de Código (Análisis SonarQube)
6. Conclusión del Proyecto

1. Contexto

Este proyecto, "Stark Industries Concurrent Security System", implementa un sistema de seguridad inteligente. Su propósito es gestionar y procesar eventos de múltiples sensores (movimiento, temperatura, acceso) en tiempo real.

La implementación se realiza utilizando FastAPI, un framework moderno de Python que facilita la creación de sistemas asíncronos y de alto rendimiento. El sistema debe garantizar una alta capacidad de respuesta, gestionar la autenticación de usuarios y notificar alertas de forma instantánea.

2. Objetivo del Proyecto

El objetivo principal es desarrollar un sistema de seguridad concurrente que cumpla con los siguientes requisitos:

* Procesar datos de múltiples sensores de forma concurrente y no bloqueante.
* Utilizar programación asíncrona (asyncio) para maximizar el rendimiento y la escalabilidad.
* Implementar un sistema robusto de autenticación y autorización basado en roles (RBAC) utilizando tokens JWT (OAuth2).
* Enviar alertas inmediatas a los clientes conectados (Dashboard) ante eventos críticos, utilizando WebSockets.
* Mantener la trazabilidad y monitorización del sistema mediante la exportación de métricas clave (Prometheus).

3. Diagnóstico Ejecutivo

El proyecto implementa con éxito un sistema de monitorización de seguridad en tiempo real altamente funcional. La arquitectura, basada en FastAPI para la API y WebSockets para la comunicación en tiempo real, se alinea perfectamente con los objetivos de concurrencia y respuesta inmediata.

El sistema presenta una arquitectura modular y desacoplada, separando claramente las responsabilidades:

* Definición de la API (main.py)
* Seguridad (security.py)
* Lógica de negocio (processing.py)
* Definición de sensores (sensors.py)
* Gestión de WebSockets (websocket.py)

Esta separación proporciona una base sólida para futuras expansiones y mantenimiento. El frontend, aunque sencillo, es completamente funcional y demuestra la viabilidad del sistema de principio a fin.

4. Análisis Técnico Detallado

A continuación, se desglosa la funcionalidad de cada módulo principal del proyecto.

4.1. main.py: Orquestador de la Aplicación y API

Este archivo es el punto de entrada que configura e inicializa la aplicación FastAPI.

* Inicialización: Configura la app, el logging básico y una instancia global del ConnectionManager para los WebSockets.
* Montaje de Rutas: Monta la ruta /static para servir el frontend y la ruta /metrics para exponer las métricas de Prometheus.
* Endpoint de Autenticación (POST /token): Maneja el inicio de sesión. Utiliza OAuth2PasswordRequestForm para recibir las credenciales, las valida contra FAKE\_USERS\_DB y genera un access\_token JWT.
* Endpoint de Eventos (POST /sensor/event): Es la ruta principal de ingesta de datos.

1. Recibe los datos y los valida contra el modelo Pydantic SensorData.
2. Verifica que el tipo de sensor exista en el SENSOR\_REGISTRY.
3. Lanza el procesamiento en segundo plano usando asyncio.create\_task(process\_sensor\_data\_concurrently(...)).
4. Responde inmediatamente con 202 Accepted para no bloquear al cliente.

* Endpoints de Sistema (Protegidos): Las rutas /system/status y /system/reset demuestran la autorización basada en roles, utilizando la dependencia Depends(get\_authorized\_user(...)).
* Endpoint de WebSocket (WS /ws/alerts): Gestiona la conexión en tiempo real. Utiliza el manager para aceptar (connect) y gestionar la desconexión (disconnect) de los clientes.

4.2. security.py: Módulo de Autenticación y Autorización

Este módulo centraliza toda la lógica de seguridad.

* Configuración: Define el SECRET\_KEY, ALGORITHM y el tiempo de expiración del token.
* Gestión de Contraseñas: Utiliza passlib.CryptContext para hashear y verificar contraseñas de forma segura.
* Definición de Roles: Un Enum (Role) define los roles ADMIN, OPERATOR y VIEWER.
* Base de Datos Simulada: FAKE\_USERS\_DB actúa como un repositorio de usuarios en memoria, almacenando contraseñas hasheadas y roles.
* Funciones JWT: create\_access\_token genera el token JWT y get\_current\_user lo decodifica y valida, extrayendo el username y role.
* Autorización (RBAC): La función get\_authorized\_user es una fábrica de dependencias. Devuelve una función (role\_verifier) que, a su vez, depende de get\_current\_user y comprueba si el rol del usuario está en la lista de roles permitidos, lanzando un HTTPException 403 FORBIDDEN si no lo está.

4.3. sensors.py: Modelos de Dominio y Lógica de Sensores

Este módulo define la estructura de los datos y la lógica de negocio de cada sensor.

* Modelos Pydantic: SensorData (entrada) y Alert (salida) garantizan la validación de tipos y la estructura de los datos que fluyen por el sistema.
* Patrón de Diseño (Strategy): Se utiliza una Clase Base Abstracta (BaseSensor) que define una interfaz común (process\_event).
* Implementaciones Concretas: MotionSensor, TemperatureSensor y AccessSensor implementan la lógica específica para decidir si un evento (data.value) debe generar una alerta CRITICAL o WARNING.
* Patrón de Diseño (Registry): SENSOR\_REGISTRY es un diccionario que mapea los nombres de los sensores a sus instancias. Esto desacopla main.py de las implementaciones concretas, permitiendo añadir nuevos sensores sin modificar la lógica de la API (similar a lo descrito en el documento de referencia).

4.4. processing.py: Gestión de Concurrencia y Lógica de Negocio

Este es el núcleo del procesamiento asíncrono del sistema.

* Gestión de Tareas Bloqueantes: Se define un ThreadPoolExecutor. La función blocking\_data\_analysis (que llama a sensor.process\_event) está diseñada para ser ejecutada en este pool de hilos.
* Orquestación Asíncrona (process\_sensor\_data\_concurrently):

1. Obtiene el bucle de eventos de asyncio.
2. Ejecuta el análisis bloqueante en un hilo separado usando loop.run\_in\_executor(executor, ...). Esto evita que el procesamiento intensivo (o cualquier I/O síncrono) bloquee el hilo principal de FastAPI.
3. Incrementa el contador de métricas events\_processed.
4. Si se genera una alerta, la difunde a todos los clientes (manager.broadcast) y lanza tareas concurrentes (asyncio.create\_task) para simular notificaciones push y de email.

4.5. websocket.py: Gestor de Conexiones en Tiempo Real

Este módulo encapsula la lógica para manejar múltiples clientes WebSocket.

* ConnectionManager: Es una clase que mantiene el estado, principalmente una lista de active\_connections.
* Ciclo de Vida: connect acepta y añade un nuevo cliente a la lista. disconnect lo elimina.
* Difusión (broadcast): Itera sobre todas las conexiones activas y envía el mensaje. Es robusto, ya que maneja excepciones (WebSocketDisconnect y Exception) para eliminar conexiones inactivas o rotas durante el proceso de difusión.

4.6. metrics.py: Monitorización y Observabilidad

Define las métricas clave para la monitorización del sistema utilizando prometheus-client.

* events\_processed (Counter): Cuenta el número total de eventos procesados.
* processing\_latency (Histogram): Mide la latencia del procesamiento de eventos, permitiendo análisis estadísticos (percentiles, etc.).
* request\_counter (Gauge): Mide el número de solicitudes HTTP activas en un momento dado.

4.7. static/: Interfaz de Usuario (Frontend)

El frontend consta de dos archivos principales que proporcionan una interfaz funcional para interactuar con el backend.

* index.html: Define la estructura del dashboard. Incluye formularios para "Autenticación" y "Simular Evento de Sensor", así como un área (<ul id="log">) para mostrar las alertas en tiempo real. Carga el script de JavaScript con el atributo defer para asegurar que el HTML se analice antes de ejecutar el script.
* script.js: Contiene toda la lógica del cliente.

1. Gestión de Estado: Almacena el accessToken en localStorage.
2. Manejador de Login: Intercepta el submit del formulario, usa fetch para enviar las credenciales a /token, almacena el token recibido y actualiza la UI.
3. Manejador de Sensores: Intercepta el submit del formulario del sensor, construye el payload JSON y lo envía a /sensor/event.
4. Gestor de WebSocket (connectWebSocket): Se conecta a /ws/alerts y define los manejadores onopen, onmessage (para añadir alertas al log), onerror y onclose (que incluye una lógica de reconexión automática cada 5 segundos).

5. Mejoras de Calidad de Código (Análisis SonarQube)

Durante el desarrollo, se identificaron y corrigieron advertencias de calidad de código, mejorando la fiabilidad del proyecto:

1. Gestión de Tareas en Segundo Plano: Se detectó que las tareas creadas con asyncio.create\_task en processing.py y main.py no se almacenaban en variables. Esto generaba un riesgo de "recolección de basura prematura" (premature garbage collection), donde la tarea podría ser eliminada por Python antes de completarse.

* Solución: Se implementó un set global (background\_tasks en processing.py) para mantener una referencia fuerte a estas tareas, y se utiliza task.add\_done\_callback(background\_tasks.discard) para limpiar el set una vez que la tarea ha finalizado.

1. Uso de datetime "Ingenuo" (Naive): En security.py, se utilizaba datetime.utcnow() para establecer el tiempo de expiración de los tokens JWT. Esta función crea objetos datetime "ingenuos" (sin zona horaria), lo cual es una práctica obsoleta y puede llevar a errores de cálculo.

* Solución: Se reemplazó datetime.utcnow() por la práctica moderna datetime.now(timezone.utc), asegurando que todos los sellos de tiempo tengan plena conciencia de su zona horaria (UTC).

6. Conclusión del Proyecto

El sistema desarrollado cumple exitosamente con todos los objetivos planteados. Se ha construido una aplicación concurrente, segura y en tiempo real utilizando tecnologías modernas como FastAPI, asyncio y WebSockets.

La arquitectura modular es limpia, mantenible y escalable. La correcta gestión de tareas bloqueantes (run\_in\_executor) y no bloqueantes (asyncio.create\_task) demuestra una sólida comprensión de la programación asíncrona en Python. El proyecto no solo funciona como se esperaba, sino que también sigue las mejores prácticas de la industria, sentando una base excelente para futuras funcionalidades.