Reto Kotlin – Sistema de Control de Aforo Inteligente en Tiempo Real

En este reto deberás diseñar y desarrollar una **app para móvil**, en **Kotlin,** un **sistema de control de aforo en tiempo real para un estadio**, capaz de procesar **eventos de entrada vía WebSocket** y **asignar automáticamente a cada asistente** el **sector y bloque óptimos** en función de su **puerta de acceso**, la **distancia física a recorrer**, la **disponibilidad de plazas** y un conjunto de **reglas de negocio dinámicas**.

El estadio está dividido en **4 sectores (Norte, Sur, Este y Oeste)**, cada uno con **100 plazas**, y a su vez en **3 bloques con capacidad fija (C, B y A)**. El sistema debe gestionar **bloqueos automáticos al alcanzar el 70% de ocupación**, **reglas especiales de acceso según el color de la camiseta**, **reasignaciones por saturación**, y calcular en tiempo real **métricas de distancia media recorrida** por bloque, por sector y de forma global.

Este ejercicio evalúa tu capacidad para **modelar dominio**, **gestionar concurrencia**, **diseñar algoritmos de decisión eficientes** y construir un **sistema robusto orientado a eventos en tiempo real**, más allá de una simple aplicación funcional.

Requisitos Android
La aplicación debe desarrollarse para **Android**, con minSdkVersion 26 (Android 8.0) y targetSdkVersion igual a la última versión estable.

Historias de usuario que describen el funcionamiento de la app.

Historia 0 – Acceso directo a la aplicación (sin login)

Como operador del estadio

Quiero acceder directamente a la aplicación sin necesidad de autenticación

Para poder supervisar el aforo y los eventos en tiempo real sin fricción ni configuración previa

---

🎯 Descripción

La aplicación **no requiere sistema de login, registro ni autenticación**.

Al iniciarse, debe mostrar directamente el **panel principal de control de aforo**, comenzando a recibir y procesar eventos en tiempo real.

✅ Escenario 0.1 – Inicio directo de la app

- **Dado** que el usuario abre la aplicación
- **Cuando** la app se inicia
- **Entonces** debe mostrarse directamente el **panel principal**
  Y el sistema debe intentar conectarse automáticamente al WebSocket
- **Y** comenzar a procesar eventos sin ninguna acción del usuario

Historia 1 - Recepción de eventos de entrada en tiempo real

Como sistema de control de aforo

Quiero recibir eventos de entrada de asistentes vía WebSocket

Para procesarlos en tiempo real y decidir su asignación o bloqueo

Escenario 1.1 – Evento válido recibido

- **Dado** que el sistema está conectado al WebSocket
- **Cuando** se recibe un evento ENTRY con puerta y color de camiseta
- **Entonces** el evento debe ser procesado exactamente una vez
- **Y** debe disparar el motor de decisión de asignación

🧩 Historia 2 - Bloqueo de acceso por camiseta multicolor

Como sistema de control de acceso

Quiero bloquear a los asistentes con camiseta multicolor

Para impedir su entrada al estadio y registrar el bloqueo

Escenario 2.1 – Camiseta multicolor

- **Dado** un evento de entrada con shirtColor = MULTICOLOR
- **Cuando** el sistema procesa el evento
- **Entonces** el asistente no debe ser asignado a ningún sector ni bloque
- **Y** debe incrementarse el contador de asistentes bloqueados

- **Y** no debe computarse ninguna distancia recorrida

🧩 Historia 3 – Asignación estándar por camino más corto

Como sistema de aforo

Quiero asignar a los asistentes al bloque más cercano disponible

Para minimizar la distancia recorrida dentro del estadio

Escenario 3.1 – Asignación al mismo sector de la puerta

- **Dado** un asistente válido que entra por la puerta Sur
- **Y** el Sector Sur tiene bloques disponibles
- **Cuando** se procesa el evento
- **Entonces** debe asignarse al bloque abierto más cercano del Sector Sur (C → B → A)

Escenario 3.2 – Sector más cercano saturado

- **Dado** que todos los bloques del Sector Sur están llenos o bloqueados
- **Cuando** entra un asistente por la puerta Sur
- **Entonces** debe asignarse al bloque válido más cercano en un sector adyacente
- **Y** la distancia total debe reflejar el cambio de sector

🧩 Historia 4 – Bloqueo automático de bloques al 70%

Como sistema de control de aforo

Quiero bloquear automáticamente los bloques al alcanzar el 70% de ocupación

Para redistribuir asistentes a los bloques menos ocupados

Escenario 4.1 – Bloque alcanza el 70%

- **Dado** un bloque con capacidad 20 y ocupación 13
- **Cuando** entra un nuevo asistente en ese bloque
- **Entonces** el bloque debe cambiar su estado a **bloqueado**
- **Y** no debe aceptar más asistentes en el futuro

Escenario 4.2 – Bloque bloqueado

- **Dado** un bloque en estado bloqueado
- **Cuando** se evalúa como posible destino
- **Entonces** debe ser descartado del algoritmo de asignación

---

🧩 Historia 5 – Regla especial de camiseta azul (sector Norte)

Como sistema de control de accesos

Quiero forzar a los asistentes con camiseta azul al Sector Norte

Para cumplir las reglas del evento

Escenario 5.1 – Sector Norte con plazas

- **Dado** un asistente con shirtColor = BLUE
- **Y** el Sector Norte tiene bloques disponibles
- **Cuando** se procesa el evento
- **Entonces** debe asignarse obligatoriamente al Sector Norte
- **Y** al bloque abierto más cercano dentro de ese sector

🧩 Historia 6 – Fallback de camiseta azul a Bloque C

Como sistema de aforo

Quiero permitir un fallback controlado para camisetas azules

Para no rechazar asistentes si el Sector Norte está lleno

Escenario 6.1 – Fallback a otro sector

- **Dado** un asistente con camiseta azul
- **Y** el Sector Norte no tiene ningún bloque válido
- **Y** existen Bloques C disponibles en otros sectores
- **Cuando** se procesa el evento
- **Entonces** debe asignarse al Bloque C más cercano de cualquier sector
- **Y** la distancia debe calcularse correctamente

Escenario 6.2 – Rechazo por falta de Bloques C

- **Dado** un asistente con camiseta azul
- **Y** no hay Bloques C disponibles en ningún sector
- **Cuando** se procesa el evento
- **Entonces** el acceso debe ser rechazado
- **Y** el rechazo debe registrarse

🧩 Historia 7 – Cálculo de distancias por asistente

Como sistema de métricas

Quiero calcular la distancia recorrida por cada asistente

Para generar métricas agregadas fiables

Escenario 7.1 – Distancia correcta

- **Dado** un asistente asignado a un bloque
- **Cuando** se completa la asignación
- **Entonces** la distancia total debe ser:
    - distancia entre sectores + distancia dentro del sector
- **Y** debe almacenarse asociada a ese bloque y sector

🧩 Historia 8 – Métricas de distancia por bloque

Como operador del estadio

Quiero ver la distancia media recorrida por bloque

Para evaluar la eficiencia del reparto

Escenario 8.1 – Cálculo de media por bloque

- **Dado** un bloque con varios asistentes asignados
- **Cuando** se calcula la métrica
- **Entonces** la distancia media debe ser la media aritmética de sus asistentes
- **Y** actualizarse tras cada nuevo evento

🧩 Historia 9 – Métricas de distancia por sector y globales

Como operador del estadio

Quiero ver métricas agregadas por sector y globales

Para entender el impacto del ruteo

Escenario 9.1 – Media por sector

- **Dado** un sector con asistentes en distintos bloques
- **Entonces** la distancia media del sector debe ser la media de todos sus asistentes

Escenario 9.2 – Media global

- **Dado** asistentes repartidos por varios sectores
- **Entonces** la distancia media global debe reflejar todos los asistentes admitidos
- **Y** no incluir a los bloqueados

🧩 Historia 10 - Visualización del estado en tiempo real

Como operador

Quiero visualizar el estado del estadio en tiempo real

Para supervisar aforo, bloqueos y métricas

Escenario 10.1 – Panel actualizado

- **Dado** un nuevo evento procesado
- **Cuando** se actualiza el estado interno
- **Entonces** el panel debe reflejar inmediatamente:
    - ocupación por sector
    - ocupación y estado por bloque
    - métricas de distancia
    - número de asistentes bloqueados

Historia 11 - Visualización y métricas del log de eventos de entrada

Como operador del estadio

Quiero ver los eventos de entrada ordenados por timestamp (del más reciente al más antiguo)

Para supervisar en tiempo real el comportamiento del sistema y el volumen de accesos aceptados y rechazados

---

🎯 Descripción

El sistema debe mantener un **log de eventos de entrada**, actualizado en tiempo real, que permita visualizar:

- Los eventos **ordenados por timestamp descendente** (más reciente arriba)
- El **resultado de cada evento** (aceptado o rechazado)
- Métricas agregadas de:
    - Total de eventos aceptados
    - Total de eventos rechazados

✅ Escenario 11.1 – Ordenación correcta del log

- **Dado** que el sistema recibe múltiples eventos de entrada
  Y los eventos pueden llegar de forma concurrente
- **Cuando** se muestran en el panel de la app
- **Entonces** los eventos deben mostrarse **ordenados por timestamp descendente**
- **Y** el evento más reciente debe aparecer siempre en la primera posición del log

---

✅ Escenario 11.2 – Identificación visual del resultado

- **Dado** un evento procesado
- **Cuando** se muestra en el log
- **Entonces** debe indicarse claramente:
    - Timestamp
    - Puerta de entrada
    - Color de camiseta
    - Resultado (ACEPTADO / RECHAZADO)
    - Sector y bloque asignado (solo si fue aceptado)

Ejemplo:

```
12:03:21 | Gate NORTE | BLUE | ACEPTADO → NORTE / C | 0m
12:03:18 | Gate OESTE | MULTICOLOR | RECHAZADO
```

✅ Escenario 11.3 – Contadores de aceptados y rechazados

- **Dado** que el sistema procesa eventos de entrada
- **Cuando** un evento es aceptado
- **Entonces** debe incrementarse el contador de eventos aceptados
- **Y cuando** un evento es rechazado
- **Entonces** debe incrementarse el contador de eventos rechazados

---

✅ Escenario 11.4 – Coherencia entre log y métricas

- **Dado** un conjunto de eventos mostrados en el log
- **Cuando** se comparan los contadores globales
- **Entonces**:
  - El número de eventos aceptados debe coincidir con los eventos ACEPTADO
  - El número de eventos rechazados debe coincidir con los eventos RECHAZADO

🧩 Historia 12 (actualizada) – Indicador de estado del WebSocket y procesamiento

Como operador del estadio

Quiero ver un indicador claro del estado del WebSocket **y del procesamiento de datos**

Para saber si la aplicación está conectada, recibiendo eventos y trabajando activamente sobre ellos

---

🎯 Descripción

La aplicación debe mostrar de forma **visible y permanente** un **indicador de estado** que refleje **dos dimensiones distintas**:

1. **Estado de conexión al WebSocket**

2. **Estado de procesamiento de eventos**

Esto evita confundir:

- “estoy conectado”
  con
- “estoy recibiendo y procesando datos”

🔁 Estados que debe soportar el indicador

El sistema debe representar **explícitamente** los siguientes estados:

1️⃣ Conectado – Inactivo

- Conexión WebSocket establecida
- **No se están recibiendo eventos**
- Estado típico al inicio o en pausas

Ejemplo visual:

- 🟢 Conectado · Sin actividad

2️⃣ Conectado – Trabajando

- Conexión WebSocket establecida
- **Se están recibiendo eventos y procesando**
- Estado normal durante operación

Ejemplo visual:

- 🟢 Conectado · Procesando eventos
- Indicador animado o pulso (opcional)

3️⃣ Reconectando

- Conexión perdida
- Reintento automático en curso
- No se reciben eventos

Ejemplo visual:

- 🟡 Reconectando…

4️⃣ Desconectado

- Sin conexión activa
- No se reciben eventos
- Datos potencialmente obsoletos

Ejemplo visual:

- 🔴 Desconectado

✅ Escenarios

Escenario 12.1 – Paso a estado “Trabajando”

- **Dado** que el WebSocket está conectado
- **Cuando** se recibe al menos un evento válido
- **Entonces** el indicador debe cambiar a estado **Conectado – Trabajando**

Escenario 12.2 – Fin de actividad

- **Dado** que el WebSocket está conectado
- **Y** no se reciben eventos durante un periodo configurable
- **Entonces** el indicador debe pasar a **Conectado – Inactivo**

Escenario 12.3 – Pérdida de conexión

- **Dado** que la aplicación estaba conectada
- **Cuando** se pierde la conexión
- **Entonces** el indicador debe pasar inmediatamente a **Desconectado**

Escenario 12.4 – Reconexión

- **Dado** que la conexión se ha perdido
- **Cuando** la app inicia el proceso de reconexión
- **Entonces** el indicador debe mostrarse como **Reconectando**
- **Y** volver a **Conectado – Inactivo** o **Trabajando** al reconectar

Servicio websocket para que el desarrollador pueda probar la app

requisitos

```shell
pip install websockets asyncio
```

Cómo ejecutar:

```shell
python websocket_server.py
```

Este servicio estará disponible: ws://localhost:8765

```python
import asyncio
import json
import random
import time
import websockets

HOST = "0.0.0.0"
PORT = 8765

GATES = ["NORTE", "SUR", "ESTE", "OESTE"]
```

```python
# Distribución de camisetas:
# - BLUE: 30%
# - MULTICOLOR: 10%
# - Otros: 60%
SHIRT_COLORS = (
    ["BLUE"] * 3 +
    ["MULTICOLOR"] * 1 +
    ["RED", "GREEN", "BLACK", "WHITE"] * 6
)

EVENT_INTERVAL_RANGE = (0.3, 1.2)  # segundos


def generate_entry_event() -> dict:
    return {
        "type": "ENTRY",
        "timestamp": int(time.time()),
        "gate": random.choice(GATES),
        "shirtColor": random.choice(SHIRT_COLORS)
    }


async def event_producer(websocket):
    try:
        while True:
            event = generate_entry_event()
            await websocket.send(json.dumps(event))
            await asyncio.sleep(random.uniform(*EVENT_INTERVAL_RANGE))
    except websockets.exceptions.ConnectionClosed:
        print("🔌 Cliente desconectado")


async def handler(websocket):
    print("✅ Cliente suscrito")
    await event_producer(websocket)


async def main():
    print(f"🚀 WebSocket server iniciado en ws://{HOST}:{PORT}")
    async with websockets.serve(handler, HOST, PORT):
        await asyncio.Future()  # run forever


if __name__ == "__main__":
    asyncio.run(main())
```