URBANTRACKER

MANUAL TÉCNICO

Sistema de gestión y visualización de rutas de transporte urbano en tiempo real

SENA

Manual técnico

Brayan Estiven Carvajal Padilla

Diego Fernando Cuellar Hernandez

Aprendices

Carlos Julio Cadena

Instructor

ANÁLISIS Y DESARROLLO DE SOFTWARE

FICHA 2900177

SERVICIO NACIONAL DE APRENDIZAJE SENA

CENTRO DE LA INDUSTRIA DE LA EMPRESA Y LOS SERVICIOS

REGIONAL HUILA

2025

Información de contacto UrbanTracker

Correo: urbantracker751@gmail.com

Contenido

1. Introducción

1.1 Propósito del Documento

Este Este manual técnico proporciona la documentación completa del sistema UrbanTracker para desarrolladores, administradores de sistemas y personal técnico encargado de la implementación, mantenimiento y soporte. El documento describe detalladamente la arquitectura, configuración, componentes y mejores prácticas del proyecto UrbanTracker.

1.2 Alcance

UrbanTracker es una plataforma web y móvil diseñada para visualizar en tiempo real la ubicación de buses del transporte público urbano[1]. El sistema abarca un servidor backend, aplicaciones móviles para conductores y usuarios, y una aplicación web de administración. Incluye funcionalidad de monitoreo de rutas, gestión de vehículos y conductores, y localización en tiempo real. Este manual cubre todos los componentes del proyecto (Backend, aplicativo móvil y web), su integración y las consideraciones técnicas para su correcto funcionamiento.

1.3 Audiencia

Este documento está dirigido a:
- Desarrolladores backend (Java/Spring) y frontend (React/React Native)
- Arquitectos de software interesados en la solución de rastreo de vehículos
- Administradores de sistemas que desplegarán y mantendrán el servidor y servicios asociados

1.4 Stack Tecnológico

UrbanTracker utiliza un stack moderno de tecnologías para lograr sus objetivos:

- **Backend:** Java 17 con Spring Boot 3.x (Spring Web, Spring Data JPA, Spring Security) – Servidor API REST y lógica de negocio[2].
- **Base de Datos:** PostgreSQL – Almacén de datos relacional para información de usuarios, rutas, vehículos, etc.[3].
- **Autenticación:** JSON Web Tokens (JWT) – Manejo de autenticación y autorización stateless en el backend[3].
- **Comunicación en Tiempo Real:** Socket.IO (WebSockets) y MQTT (Eclipse Mosquitto) – Para transmisión de ubicaciones de buses en vivo[4][3]. El sistema aprovecha Socket.IO para notificaciones web y MQTT para actualización eficiente desde los móviles.
- **Frontend Web (Administración):** Next.js (React 18 + TypeScript) – Aplicación web para panel de control administrativo. Usa Tailwind CSS para diseño responsivo e Iconos Lucide para gráficos[5].

- **Aplicación Móvil:** React Native con Expo (SDK 49+) – Aplicaciones móviles multiplataforma. Se desarrolla una aplicación Expo: para **conductores** (envío de ubicación). En TypeScript utilizando Expo Router para la navegación.
- **Servicios de Mapas:** API de MapBox y SDK de MapBox – Integración de mapas en las apps para mostrar rutas y posiciones.
- **Otros:** Herramientas de diseño de UI como Figma para prototipos; Git para control de versiones colaborativo[6].

Cada tecnología se usa con un propósito específico dentro de la solución (ver sección 2 Arquitectura). Este stack tecnológico permite una plataforma escalable, en tiempo real.

2. Arquitectura del Si3tema

2.1 Arquitectura General

UrbanTracker implementa una arquitectura **multicomponente** de tipo cliente-servidor. Los principales componentes son:

- **Servidor Backend (API REST):** Aplicación **Spring Boot** desplegada en servidor que expone endpoints REST y **orquesta casos de uso del dominio**. Provee **autenticación JWT**, persistencia en **PostgreSQL** y publica/consume mensajes (p. ej., **MQTT/eventos**) para actualizaciones en tiempo real. Sigue una **arquitectura Hexagonal con enfoque DDD y organización por feature**: cada módulo de negocio (p. ej., `vehicles`, `routes`, `journeys`) se divide en **`domain`** (entidades/agregados, *value objects* y **ports** de repositorio, sin dependencias de framework), **`application`** (servicios de aplicación/casos de uso, **DTOs** de request/response y mapeos **DTO↔Dominio**), **`infrastructure`** (adaptadores de entrada como **controladores REST** y de salida como **persistencia** e integraciones externas; incluye **modelos JPA** y mapeos **Dominio↔Persistencia**) y **`repository`** (implementaciones de los **ports** usando **Spring Data JPA**, p. ej., `repository/impl` + `repository/jpa`). El paquete **`exception`** concentra las excepciones del módulo y su mapeo a respuestas HTTP. Este diseño **desacopla el núcleo de negocio de la infraestructura**, mejora la **testabilidad** y deja al proyecto **listo para escalar o extraer módulos como microservicios**.

- **Aplicación Web de Administración:** Interfaz web para administradores del sistema, desarrollada en Next.js/React. Permite la gestión de rutas, vehículos, conductores y visualización en un panel de control. Se comunica con el backend a través de las API REST y muestra datos en tiempo real (por ejemplo, posición actual de buses) mediante WebSockets. Incluye componentes de UI como barras laterales, dashboards y mapas integrados.

- **Aplicación Móvil de Conductor:** Aplicación React Native separada para los conductores o buses que no cuentan con GPS dedicado. Esta app captura la ubicación GPS del dispositivo móvil del conductor y la envía periódicamente al sistema (a través de MQTT al broker Mosquitto, por ejemplo). De esta forma, la posición del bus se actualiza en la plataforma en tiempo real[4]. La app de conductor funciona en segundo plano (con permisos especiales de ubicación) y garantiza el envío continuo de coordenadas.

- **Broker de Mensajes en Tiempo Real:** UrbanTracker puede usar un broker MQTT (Eclipse Mosquitto) para canalizar las posiciones enviadas por los móviles de conductor. Alternativa o adicionalmente, el backend expone endpoints de WebSocket (usando Socket.IO) para notificar a los clientes web de usuario sobre cambios de ubicación. Esta arquitectura de mensajería garantiza actualizaciones instantáneas con baja latencia.

- **Base de Datos Central:** PostgreSQL almacena la información maestra: usuarios, roles, compañías de transporte, vehículos, rutas predefinidas y puntos de ruta. La persistencia se maneja vía JPA/Hibernate en el backend, manteniendo un modelo relacional consistente.

Estos componentes operan para brindar una plataforma unificada: los conductores emiten su posición, el backend consolida y distribuye, y los administradores/pasajeros consumen la información actualizada. La comunicación principal sigue un patrón **cliente-servidor** sobre HTTP/HTTPS para las operaciones REST tradicionales, complementado con **eventos en tiempo real** via websockets/MQTT para la funcionalidad crítica de tracking en vivo.

2.2 Patrón de Diseño y Capas

El backend de UrbanTracker emplea una arquitectura por capas siguiendo el patrón MVC (Modelo-Vista-Controlador) adaptado a servicios web RESTful, con una clara separación de responsabilidades. Adicionalmente, se aprovechan otros patrones de diseño, destacando el patrón **Fábrica Abstracta** (*Abstract Factory*) para desacoplar la creación y obtención de objetos de la lógica de negocio. Las principales capas del sistema son:

- **Capa de Controladores (Controllers):** Define los endpoints REST (rutas HTTP) agrupados por contexto (usuarios, vehículos, rutas, etc.). Los controladores reciben las solicitudes HTTP, validan los datos de entrada y delegan la lógica a la capa de servicio. Por ejemplo, el `UserController` maneja rutas como `/api/v1/public/user` para operaciones sobre usuarios.
- **Capa de Servicio (Services):** Implementa la lógica de negocio y coordina las operaciones. Cada entidad principal posee un servicio dedicado (por ejemplo, `UserService`, `VehicleService`, etc.) que encapsula las reglas de negocio y la orquestación de datos. Los servicios interactúan con los repositorios para

acceder a la base de datos y construyen las respuestas (o DTOs) que serán retornadas al controlador.

- **Capa de Datos (Repositories y Modelos):** Se encarga de la persistencia, abstrayendo la comunicación con la base de datos (PostgreSQL) mediante Spring Data JPA. Por cada entidad se define una interfaz de repositorio (p. ej., IUser, IRoute, IVehicle) que extiende JpaRepository, proporcionando métodos CRUD básicos y permitiendo consultas JPQL personalizadas. Los modelos (entidades) son clases Java anotadas con JPA (@Entity) que mapean a tablas de la base de datos (por ejemplo, User, Vehicle, Company). Cabe destacar que este enfoque implementa el patrón **Abstract Factory**: Spring genera implementaciones concretas de las interfaces de repositorio en tiempo de ejecución, actuando como fábricas que proveen instancias de entidades gestionadas sin exponer los detalles de su creación o acceso a datos a las capas superiores.
- **Capa de Seguridad:** Configurada mediante Spring Security, incluye filtros JWT personalizados, un servicio de autenticación y la integración de la entidad de usuario con el mecanismo de seguridad de Spring (la clase User implementa UserDetails para este propósito). Esta capa transversal se encarga de validar credenciales y proteger los endpoints de la aplicación (ver sección 7 para más detalles).

En el frontend web se sigue una arquitectura de componentes y enrutamiento provisto por Next.js. La interfaz se estructura en componentes reutilizables (ej. Sidebar, Dashboard) y páginas que corresponden a vistas o secciones de la aplicación. Next.js utiliza el App Router con segmentos anidados para organizar las páginas (p. ej., `/Dashboard/vehicles` para la sección de vehículos). Se aplica el principio de separar la lógica de datos (llamados a la API, hooks de datos) de la presentación (componentes de UI), facilitando mantenibilidad.

La aplicación móvil sigue un patrón similar a **MVVM** simplificado: - **Vistas (Screens)** definidas como componentes React Native (JSX) para cada pantalla de la app. **Modelo de Vista (Providers/Context)** que usa Context API de React para manejar estado global (p. ej., AuthProvider para estado de autenticación, LocationProvider para datos de GPS). Los providers encapsulan la lógica como obtener token, verificar sesión, enviar ubicación, etc. - **Servicios/API**: módulos que encapsulan llamadas a las APIs REST. Ejemplo: `authService.login()` en la app del conductor simula o realizará peticiones de login[12]. Esta capa aísla los detalles de comunicación (fetch/axios) del resto de la app. - **Motores de Navegación:** Expo Router organiza la navegación en las apps móviles de forma declarativa (archivos en `src/app` definen rutas). Esto impone un orden claro en la estructura de pantallas y facilita la navegación tipo stack o tab según necesidad.

2.3 Principios de Diseño

UrbanTracker fue diseñado con los siguientes principios en mente: - **Modularidad:** Cada funcionalidad (usuarios, vehículos, rutas, tracking) está modularizada. En el

backend, esto se refleja en paquetes separados (`controller`, `service`, `model`, `repository`), y en el frontend, en componentes y contextos independientes. Esto facilita futuras ampliaciones (por ejemplo, agregar un módulo de notificaciones) sin impactar otras partes. - **Escalabilidad:** El uso de mensajería en tiempo real desacopla la generación y consumo de eventos de ubicación, permitiendo escalar componentes por separado. Por ejemplo, podría haber múltiples instancias del backend suscritas al broker MQTT para distribuir la carga de procesamiento de ubicaciones. - **Mantenibilidad:** Se aplican convenciones de código consistentes (nombrado de clases `I<Entidad>` para repos, DTOs separados, uso de Lombok para reducir boilerplate). La separación de capas permite realizar cambios (p. ej. cambiar de PostgreSQL a otro SGBD, o de MQTT a otro protocolo) con impacto mínimo en el resto del sistema. - **Seguridad y Robustez:** La arquitectura incorpora seguridad desde el diseño (JWT, roles) y manejo de errores centralizado. Se han añadido validaciones exhaustivas en servicios (ej. formato de datos, duplicados) antes de persistir[13][14]. Asimismo, se contemplan mecanismos de recuperación ante fallos de conexión (reintentos o logs de error) para garantizar la estabilidad del sistema. - **Experiencia de Usuario en Tiempo Real:** La elección de WebSockets/MQTT y la arquitectura orientada a eventos responden al principal requerimiento de UrbanTracker: presentar información en **tiempo real**. La arquitectura evita polling intensivo y prefiere notificaciones push al cliente, optimizando la experiencia (las ubicaciones aparecen al instante) y el uso de recursos de red.

En resumen, UrbanTracker se construye sobre una base sólida de múltiples capas y servicios especializados, asegurando que el sistema sea **flexible**, **escalable** y **fácil de mantener**, a la vez que cumple con la funcionalidad crítica de rastreo en vivo de manera eficiente.

3. Requisitos Técnicos

3.1 Requisitos de Software

Para desplegar o desarrollar UrbanTracker se necesita el siguiente software instalado en los entornos correspondientes:

- **JDK (Java Development Kit) 17 o superior:** Necesario para compilar y ejecutar el backend Spring Boot.
- **Apache Maven 3.8+**: Herramienta de construcción utilizada para gestionar las dependencias y compilar el proyecto backend[15].
- **PostgreSQL 14 o superior:** Sistema de gestión de base de datos relacional donde residirán los datos de la aplicación (puede estar instalado localmente para desarrollo, o en un servidor para producción).
- **Node.js 18+ y NPM:** Entorno de ejecución JavaScript para construir y ejecutar las aplicaciones web y móvil. Se recomienda Node 18 LTS. NPM (o Yarn) se usa para instalar dependencias de los proyectos React/React Native[15].

- **Expo CLI:** (Opcional) Herramienta de línea de comando de Expo para ejecutar y construir las aplicaciones móviles. Se instala via NPM (`npm install -g expo-cli`).
- **IDE/Editors recomendados:** IntelliJ IDEA o Eclipse (para backend Java), Visual Studio Code (para frontend web/móvil).
- **Herramientas de Mapas:** (Opcional según necesidades) Para funcionalidades de mapas en desarrollo, puede requerirse configurar credenciales de API Mapbox.
- **Herramientas de Pruebas:** Postman o similar para probar endpoints de la API REST; Expo Go (app móvil) para probar las apps en dispositivos sin compilar; Android Studio (emulador Android) para pruebas locales de la app móvil.

3.2 Requisitos de Hardware y Dispositivos

- **Servidor Backend:** Se recomienda un servidor con al menos 2 CPU y 2 GB de RAM para entornos de prueba/desarrollo. En producción, dimensionar según carga esperada (el consumo principal de RAM vendrá de la JVM de Spring Boot y de la base de datos). Almacenamiento de ~5 GB es suficiente para iniciar (la base de datos crecerá con registros históricos de viajes; considerar ampliación según retención de datos).
- **Base de Datos:** Si el DB server está separado, asegurar al menos 1 CPU, 1 GB RAM dedicados para PostgreSQL para un desempeño adecuado en un inicio. El almacenamiento debe soportar el crecimiento en el número de registros de localización si se guardan trayectos históricos (opcional).
- **Broker MQTT (Mosquitto):** Si se despliega un broker para producción, su uso de recursos es bajo; una instancia pequeña puede manejar cientos de clientes. Puede correr en el mismo servidor del backend o separado por aislamiento.
- **Clientes Web:** UrbanTracker Web es una aplicación ligera que corre en el navegador; se soportan los navegadores modernos (Chrome, Firefox, Edge) en sus versiones recientes. No requiere hardware especial del lado cliente, más allá de capacidad para ejecutar un navegador moderno.
- **Dispositivos Móviles Compatibles:** Las aplicaciones móviles están enfocadas a **Android**. Se han probado en Android 10 y superior. Requisitos mínimos: dispositivo con GPS, 2 GB de RAM, Android 8.0+ (SDK 26) o superior. La app de pasajeros requiere conectividad a Internet para recibir actualizaciones; la app de conductor requiere GPS y conexión activa. *Nota:* Aunque Expo permite generar también una app iOS, inicialmente UrbanTracker está orientado a Android (no se ha testeado en iOS).
- **Emulador/Simulador:** Para desarrollo móvil, se puede usar el Emulador de Android Studio (AVD) con imagen x86 de Android 11+. Asegurarse de habilitar las *Google APIs* si se necesita Google Maps en las pruebas.

En general, el sistema no demanda hardware especializado: funciona en infraestructura estándar x64 para el servidor y en smartphones Android comerciales para la parte móvil. Es importante contar con buena conectividad de datos en los

dispositivos de conductor para el envío en tiempo real, y en los dispositivos de usuario para la recepción.

4. Instalación y Configuración

A continuación, se describen los pasos para instalar y configurar el entorno de UrbanTracker en modo de desarrollo local. Se asume que se cuenta con los requisitos de software mencionados en la sección 3.1.

4.1 Preparación del Entorno

1. **Clonar el repositorio:** Obtener el código fuente desde GitHub. En una terminal, ejecutar:

```
git clone https://github.com/AFSB114/UrbanTracker.git
cd UrbanTracker
```

Esto descargará el proyecto completo y ubicará la terminal en el directorio raíz del proyecto[16].

1. **Instalar dependencias del Backend:** Navegar al directorio `Backend/` y ejecutar Maven para descargar las dependencias definidas en el pom.xml.

   ```
   cd Backend
   mvn clean install
   ```

   Esto compilará el código Java y asegurará que todas las librerías (Spring, JWT, etc.) estén disponibles en el repositorio local de Maven.

2. **Instalar dependencias del Frontend Web:** En otra terminal, ir al directorio `Web-Admin/` (aplicación web de administración) y ejecutar:

   ```
   npm install
   ```

   Esto descargará los paquetes npm necesarios (React, Next.js, Tailwind, etc.). Hacer lo mismo en el directorio `Web-Client/` si existe o si la app web fue refactorizada bajo otro nombre (en la rama dev puede aparecer como `Web-Client`).

3. **Instalar dependencias de las Apps Móviles:** Similarmente, entrar en `Movil-User-Client/` y ejecutar `npm install`, luego en `Movil-Driver-Client/` ejecutar `npm install`. Esto instalará los paquetes Expo, React Native, etc., necesarios para cada aplicación móvil.

4.2 Configuración de la Base de Datos

Antes de ejecutar el backend, se debe configurar la conexión a PostgreSQL:

- **Crear la base de datos:** Iniciar PostgreSQL y crear una BD vacía, por ejemplo, llamada `urbantracker_db`. También crear un usuario con permisos (por ejemplo, usuario `urban_admin` con contraseña `urban_pass`). Esto puede hacerse via línea SQL:

```
CREATE DATABASE urbantracker_db;
CREATE USER urban_admin WITH ENCRYPTED PASSWORD 'urban_pass';
GRANT ALL PRIVILEGES ON DATABASE urbantracker_db TO urban_admin;
```

- **Configurar credenciales:** En el backend, el archivo de propiedades de Spring Boot (`application.properties` o `application.yml`) debe contener la URL de conexión y las credenciales. Por ejemplo, editar `Backend/src/main/resources/application.properties` con:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/urbantracker
_db
spring.datasource.username=urban_admin
spring.datasource.password=urban_pass
spring.jpa.hibernate.ddl-auto=update
```

  Este último parámetro (`ddl-auto=update`) hará que JPA cree o actualice las tablas automáticamente según las Entities definidas. Asegurar que el puerto, nombre de host, usuario y contraseña correspondan a su instalación de PostgreSQL.

- **Datos iniciales:** Opcionalmente, se pueden precargar datos básicos. Por ejemplo, insertar roles (`ADMIN`, `DRIVER`) y quizás un usuario admin inicial. Esto se puede hacer manualmente con sentencias SQL o creando un *data.sql* que Spring Boot ejecute al arrancar. *Ejemplo:*

```
INSERT INTO role(name) VALUES ('ADMIN'), ('DRIVER');
INSERT INTO users(username, password, role_id) VALUES ('admin',
'admin123', 1);
```

  Nota: La contraseña idealmente debe estar encriptada (ver sección 7 Seguridad). Para una configuración rápida de desarrollo, se puede almacenar texto plano, pero **no es recomendado en producción**.

4.3 Ejecución del Servidor y Clientes

Una vez configurados los puntos anteriores, proceder a ejecutar cada componente:

- **Ejecución del Backend:** Desde el directorio `Backend/`, iniciar la aplicación Spring Boot. En desarrollo, la forma más sencilla es:

  ```
  mvn spring-boot:run
  ```

  Esto lanzará el servidor en `http://localhost:8080` (puerto por defecto de Spring Boot, a menos que se haya cambiado). En la consola, asegurarse de ver mensajes de arranque exitoso de Tomcat y la conexión a PostgreSQL. El backend expone la API REST y la interfaz de documentación Swagger (ver sección 8.1).

- **Ejecución de la App Web (Administración):** Desde `Web-Admin/`, iniciar el entorno de desarrollo de Next.js:

  ```
  npm run dev
  ```

  Esto arrancará un servidor de desarrollo (por defecto en `http://localhost:3000`). Abrir ese URL en un navegador para cargar la aplicación web. Como es la interfaz administrativa, es posible que primero muestre una pantalla de login (pendiente de implementar en su totalidad) o acceso directo al dashboard si no hay autenticación en la web aún. Asegúrese de tener el backend levantado para que las API funcionen. Durante desarrollo, la app web se recarga automáticamente ante cambios de código (hot-reload).

- **Ejecución de la App Móvil de Usuario:** Navegar a `Movil-User-Client/` y ejecutar:

  ```
  expo start
  ```

  Esto abrirá la interfaz de Expo en la terminal y posiblemente en el navegador. Desde allí, escanear el código QR con la aplicación Expo Go en un dispositivo Android **o** pulsar la opción de ejecutar en emulador Android (asegurarse de tener uno abierto). La app se cargará en modo desarrollo. Para que la app móvil se comunique con el backend local, se debe configurar la URL base de la API en la app (por ejemplo, usando `Constants.manifest.debuggerHost` de Expo para obtener la IP local). En desarrollo, Expo suele proxyear o permitir tunel, pero es recomendable configurar manualmente la IP del PC en la app móvil para las peticiones (por ejemplo `http://192.168.1.100:8080` en lugar de localhost, ya que *localhost dentro del móvil no es el mismo host que su PC*). Esto puede hacerse en un archivo de configuración o directamente en el código de servicio API.

- **Ejecución de la App Móvil de Conductor:** Análogamente, entrar a `Movil-Driver-Client/` y ejecutar `expo start`. Esta app mostrará su propia interfaz

(principalmente centrada en login del conductor y pantalla de mapa/ubicación). Tras iniciar, se deben conceder permisos de geolocalización. En expo, se solicitarán en la primera ejecución. La app en modo dev mostrará logs en la consola de Metro bundler. Asegúrese también de configurar la URL del backend/MQTT broker en esta app si es necesario (por ejemplo, el broker MQTT podría estar en la misma máquina, use la IP local).

Configuración de MQTT (opcional): Si se va a usar MQTT para el flujo de ubicación, se debe tener una instancia de Mosquitto corriendo. En desarrollo, puede instalar Mosquitto localmente y ejecutarlo (por defecto en el puerto 1883). Configure la app de conductor para publicar en la dirección IP donde esté el broker (puede ser el mismo servidor backend o `localhost` si el broker corre en el PC y la app se conecta via la red local). De igual modo, configure el backend o la app de usuario para suscribirse a los tópicos necesarios. Si la configuración de MQTT no está lista, el sistema puede funcionar en desarrollo simplemente enviando las ubicaciones a través de llamadas HTTP periódicas (p. ej., un endpoint REST) o simulando los movimientos.

Variables de Entorno adicionales: Asegúrese de establecer cualquier variable de entorno que el sistema requiera. Por ejemplo: - `JWT_SECRET` (si en lugar de llave fija se quisiera definir externamente la clave secreta JWT), - `EXPO_GOOGLE_MAPS_API_KEY` para que Expo utilice Google Maps (si aplica), - Variables de configuración de rutas de mapas o de servicios externos.

Estas variables pueden definirse en un archivo `.env` en los proyectos front-end (Next.js y Expo soportan cargar de `.env` variables) o configurarse en el entorno antes de lanzar las aplicaciones.

Con estos pasos, el entorno de UrbanTracker debería estar en marcha: el backend atendiendo en el puerto 8080, la aplicación web accesible en el puerto 3000, y las apps móviles conectándose (vía Expo) al backend. Se recomienda probar rápidamente un flujo básico: crear un usuario mediante la API o base de datos, luego intentar login (si implementado) o simular datos, y verificar en la consola logs de actividad.

5. Estructura del Proyecto

El repositorio UrbanTracker está organizado en múltiples carpetas en la raíz, separando claramente cada subproyecto y componente del sistema. A continuación, se presenta una vista simplificada del árbol de directorios y una breve descripción de cada parte principal:

```
UrbanTracker/
├── backend/
```

```
|   ├── src/
|   |   ├── main/java/com/sena/urbantracker/
|   |   |   ├── vehicles/                 # Módulo de negocio (feature): Vehículos
|   |   |   |   ├── domain/               # Núcleo DDD (sin framework)
|   |   |   |   |   ├── entity/           # Entidades/Aggregates del dominio
|   |   |   |   |   ├── valueobject/      # Value Objects con invariantes
|   |   |   |   |   └── repository/       # Ports (interfaces) del dominio
|   |   |   |   ├── application/          # Casos de uso (Servicios de aplicación)
|   |   |   |   |   ├── service/          # Orquestación de reglas + transacciones
|   |   |   |   |   ├── dto/              # Contratos de entrada/salida
|   |   |   |   |   |   ├── request/
|   |   |   |   |   |   └── response/
|   |   |   |   |   └── mapper/           # Mapeo DTO ↔ Dominio
|   |   |   |   ├── infrastructure/       # Adapters (entrada/salida)
|   |   |   |   |   ├── controller/       # REST Controllers (inbound adapter)
|   |   |   |   |   ├── external/         # Integraciones externas (MQTT, APIs), si aplica
|   |   |   |   |   └── persistence/      # Persistencia (outbound adapter)
|   |   |   |   |       ├── model/        # Modelos JPA (solo infraestructura)
|   |   |   |   |       └── mapper/       # Mapeo Dominio ↔ Persistencia
|   |   |   |   |   ├── repository/       # Implementaciones técnicas de Ports
|   |   |   |   |   |   ├── jpa/          # Spring Data JPA repositories (sobre *persistence.model*)
|   |   |   |   |   |   └── impl/         # Adapter que implementa Ports con JPA
```

```
│   │   │   │   └── exception/            # Excepciones del módulo
(mapeables a HTTP)
│   │   │   ├── routes/                   # Módulo de Rutas (misma
estructura que vehicles)
│   │   │   ├── journeys/                 # Módulo de Viajes/Asignaciones
(misma estructura)
│   │   │   ├── companies/                # Módulo de Empresas/Operadores
(misma estructura)
│   │   │   ├── users/                    # Módulo de Usuarios/Roles/Auth
(misma estructura)
│   │   │   └── shared/                   # Cross-cutting y componentes
compartidos
│   │   │       ├── config/               # Config general
(Swagger/OpenAPI, CORS, etc.)
│   │   │       ├── security/             # Seguridad (JWT
filter/service, Resource Server)
│   │   │       ├── exception/            # Handler global de errores
(ControllerAdvice)
│   │   │       └── util/                 # Utilidades comunes
(constantes, helpers)
│   │   ├── main/resources/
│   │   │   ├── application.properties    # Config de base de datos,
seguridad, app
│   │   │   └── … (otros recursos)
│   │   └── test/java/com/sena/urbantracker/
│   │       └── …                         # Pruebas por módulo (dominio,
app, adapters)
│   ├── pom.xml backend)
│   └── ... (archivos adicionales de construcción)
├── Web-Admin/
│   ├── src/
│   │   ├── app/                # Estructura de páginas Next.js (rutas de
la aplicación web)
│   │   ├── components/         # Componentes reutilizables (UI,
gráficos, mapas)
│   │   └── lib/                # Utilidades (hooks, helpers)
│   ├── public/                 # Recursos públicos (imágenes, favicon,
etc.)
```

```
│   ├── next.config.ts          # Configuración de Next.js
│   ├── tailwind.config.js      # Configuración Tailwind CSS
│   └── package.json            # Dependencias del proyecto web
├── Web-Client/ (en dev puede existir esta carpeta si la app web se
renombró)
│   └── ... (estructura similar a Web-Admin, si aplica)
├── Movil-User-Client/
│   ├── src/
│   │   ├── app/                # Screens y navegación (Expo Router) para
la app de usuario
│   │   ├── components/         # Componentes UI (p. ej., mapas, botones)
│   │   ├── map/                # Integración de mapas (MapBox, Google)
│   │   ├── providers/          # Context Providers (ej: contexto de
autenticación, ubicación)
│   │   └── services/           # Servicios API para consumir backend
│   ├── assets/                 # Imágenes, íconos (logos, splash)
│   ├── App.tsx                 # Punto de entrada de la app (registra el
NavigationContainer)[17]
│   ├── app.json                # Configuración de la app Expo (nombre,
permisos)
│   └── package.json            # Dependencias de la app móvil de usuario
├── Movil-Driver-Client/
│   ├── src/
│   │   ├── app/                # Screens y navegación de la app de
conductor (login, mapa)
│   │   ├── components/         # Componentes UI (similar estructura a
user)
│   │   ├── providers/          # Contexto de ubicación, auth para
conductor
│   │   └── services/           # Servicios (p. ej., envío de ubicación,
autenticación)
│   ├── assets/                 # Recursos (logos, iconos)
│   ├── App.tsx                 # Punto de entrada de la app de conductor
│   ├── app.json                # Config Expo (incluye permisos GPS en
Android)[18]
│   └── package.json            # Dependencias de la app móvil de
conductor
├── README.md                    # Documentación general del proyecto (en
GitHub)
└── manual-tecnico-schoolme.pdf  # *Ejemplo de manual técnico
proporcionado*
```

Algunos puntos a destacar de la estructura:

Se **separan claramente los dominios**: backend contiene el servidor; cada cliente (web, móvil) tiene su propio directorio, habilitando trabajo en paralelo por equipos.

Dentro de backend/src/main/java/com/sena/urbantracker/, los paquetes se **organizan por feature** y siguen **Arquitectura Hexagonal + DDD**: domain (reglas y Ports), application

(use cases/DTOs), infrastructure (adapters REST/DB/brokers) y repository (impls técnicas de Ports con JPA/Abstract Factory). Esto aporta **desacoplamiento**, **testabilidad** y **escalabilidad** futura (microservicios por módulo).

- Las aplicaciones Expo (`Movil-User-Client` y `Movil-Driver-Client`) comparten una estructura similar, lo que estandariza el desarrollo móvil. Cada una tiene su **router** (en `src/app`) y puede compartir componentes o lógica, pero se mantienen separadas ya que las funcionalidades de un pasajero y un conductor difieren. - Existe una carpeta `Web-Client` y `Web-Admin`; esto indica que durante el desarrollo se pudo haber comenzado con un nombre genérico (*Web-Client*) y luego enfocado hacia *Web-Admin* como panel administrativo. En la rama `dev`, asegurarse de cuál es la aplicación web activa.

En la actualidad **Web-Admin** es la que contiene el Dashboard y componentes como Sidebar, etc., mientras que *Web-Client* podría ser código legado o experimental[20][21]. - Los archivos de configuración esenciales (por ejemplo, `application.properties` para backend, `tailwind.config.js` para estilos, `app.json` para Expo) están presentes en las ubicaciones estándar, listos para ser modificados según el entorno (por ejemplo, en producción el `application.properties` debería apuntar a la BD y URLs de producción, etc.). - El repositorio incluye documentación de apoyo y scripts. Por ejemplo, se ve un archivo de troubleshooting para ubicación en la app de conductor (`TROUBLESHOOT_LOCATION.md`), lo cual es útil para consultar soluciones a problemas comunes durante desarrollo[22]. También es posible que existan diagramas UML o documentación en carpetas especiales (e.g., `docs/`) si el equipo las agrega.

En resumen, la estructura del proyecto es **coherente y bien organizada**, permitiendo localizar rápidamente la implementación de una funcionalidad específica. Por ejemplo, si se desea modificar cómo se listan los vehículos, se sabe que habrá que tocar `VehicleController` (controlador), `VehicleService` (lógica) y posiblemente la entidad `Vehicle` o el repo `IVehicle`.

6. Ge3tión de Dependencia3

UrbanTracker utiliza herramientas de gestión de dependencias para asegurar la consistencia de versiones y facilitar la construcción del software:

6.1 Backend (Maven):

El proyecto backend es un módulo Maven. Todas las dependencias externas (frameworks y librerías Java) están declaradas en el archivo `pom.xml`[23]. Maven descarga automáticamente estas librerías desde repositorios centralizados (Maven Central u otros) cuando se compila el proyecto. Algunas dependencias clave en el pom.xml incluyen: - *spring-boot-starter-data-jpa*: driver de JPA/Hibernate para interacción con la base de datos[24].

- *spring-boot-starter-web*: para crear la API REST (contiene Tomcat embebido, JSON Jackson, etc.)[24].
- *spring-boot-starter-security*: integración de Spring Security para autenticación/autorización[25].
- *postgresql*: controlador JDBC de PostgreSQL para permitir a Hibernate conectarse a la base de datos[26].
- *jjwt (io.jsonwebtoken)*: librería para la creación y validación de JWTs[27].
- *springdoc-openapi-ui*: para la generación automática de documentación Swagger UI de la API[28].
- *lombok*: librería para reducir código repetitivo (getters/setters, constructores) en las clases modelo[29].

Maven se encarga de manejar las versiones transitivas (por ejemplo, Spring Boot Starter trae varias dependencias internas). En el `<dependencyManagement>` de Spring Boot se fija una versión base (en este caso Spring Boot 3.5.3, Java 17) que armoniza todas las sub-dependencias. Si se requiere agregar una librería nueva (por ejemplo, una librería de envío de correos), se edita el pom.xml agregando la dependencia con su versión. Tras ello, Maven resolverá y la próxima compilación descargará dicha librería.

La gestión de **versiones** se controla en el pom; por ejemplo, la versión de Java se especifica en `<java.version>17</java.version>`[30]. Para actualizar versiones (p.ej. migrar a Spring Boot 3.6 en un futuro), se ajusta la versión del parent y se verifica compatibilidad.

6.2 Frontend Web y Móvil (Node.js/NPM):
En los proyectos web y móviles, las dependencias se manejan con **npm** (Node Package Manager) mediante archivos `package.json` en cada subproyecto. Cada package.json lista los paquetes requeridos y sus versiones. Por ejemplo, en la app web (Web-Admin) se incluyen dependencias como React, Next.js, Tailwind, etc., mientras que en las apps móviles (Expo) se listan react-native, expo-modules, react-navigation, etc.

Cuando se ejecuta `npm install`, NPM lee el package.json y descarga las versiones especificadas de cada paquete, almacenándolas en la carpeta `node_modules`. Además, se genera/actualiza un archivo de bloqueo de versiones (`package-lock.json` o `yarn.lock` si se usara Yarn) para asegurar que todos los desarrolladores instalen exactamente las mismas versiones, evitando **discrepancias de entorno**.

Algunas dependencias destacadas en los frontends: - **Web-Admin:** `next`, `react`, `react-dom`, `@tailwindcss`, `lucide-react` (icons), posiblemente `axios` o fetch polyfills para llamadas API, etc. - **Expo Apps:** `react-native`, `expo`, `expo-router`, `@react-native-async-storage/async-storage` (para almacenar tokens), `expo-location` (para obtener GPS), `mqtt` o `socket.io-client` (si se usa en cliente para tiempo real), entre otras.

Para agregar un nuevo paquete (ej: una librería de gráficos) en el frontend se usa `npm install <package>`. Esto actualizará el package.json y descargará el paquete. Es importante commitear los cambios en package.json y lock file para que otros puedan replicarlo.

Gestión de Dependencias Comunes: No hay un monorepo unificado con un solo package.json para todo, sino que cada proyecto mantiene las suyas. Esto evita conflictos de versiones entre, por ejemplo, el proyecto web y el móvil. Cabe destacar que se debe manejar con cuidado las versiones de React entre web y móvil; aunque son entornos distintos, mantener React Native y React sincronizados en versiones compatibles ayuda a los desarrolladores (por ejemplo, RN 0.72 viene con React 18).

Plugins y Configuraciones: Tanto Maven como Expo/Next utilizan plugins de apoyo. En Maven se ha configurado el plugin maven-compiler con soporte para Lombok y el plugin spring-boot-maven para empaquetar la aplicación fácilmente[31][32]. En Expo, en el app.json se ve el uso de plugins de Expo como *expo-location* (para permisos de GPS) y *expo-router*[18][33]. Estos plugins también son una forma de dependencia, manejados via configuraciones en JSON en lugar de en package.json directamente, pero Expo los instala como paquetes.

En resumen, la gestión de dependencias de UrbanTracker está automatizada: Maven para backend y NPM para frontends. Se recomienda: - Actualizar periódicamente las dependencias a versiones estables más recientes (especialmente por parches de seguridad). - Revisar los changelogs de Spring Boot, Expo, etc., antes de saltar de versión para anticipar cambios incompatibles. - Usar versiones fijas (evitar rangos flotantes) para garantizar reproducibilidad del build.

7. Autenticación y Seguridad

La plataforma implementa varias capas de seguridad para proteger el acceso y los datos sensibles. A continuación, se detallan los mecanismos de autenticación, manejo de tokens, roles de usuario y prácticas de seguridad adoptadas.

7.1 Sistema de Autenticación (JWT)

UrbanTracker utiliza **JSON Web Tokens (JWT)** para autenticar a los usuarios de sus aplicaciones (administradores y conductores, principalmente). El flujo de autenticación es el siguiente:

1. **Inicio de Sesión (Login):** Un usuario (p. ej., un administrador en la web o un conductor en la app móvil) ingresa sus credenciales (nombre de usuario/contraseña) en la aplicación. La aplicación envía estas credenciales al endpoint de login del backend (por ejemplo, `POST /api/v1/auth/login`). *Nota:* En la rama actual, este endpoint puede implementarse dentro de `UserController` o mediante la configuración de Spring Security. En una

implementación típica, un AuthenticationManager de Spring valida las credenciales contra la base de datos.

2. **Validación Backend:** El servidor verifica las credenciales recibidas. Si son correctas (el usuario existe y la contraseña coincide, tras aplicar hash BCrypt), se procede a generar un JWT. Si son incorrectas, se devuelve un error 401 Unauthorized.

3. **Generación de JWT:** El token JWT se crea incorporando en sus *claims* datos del usuario como: identificador de usuario, nombre de usuario y rol. Por ejemplo, al generar el token se inserta el `user.id` y el `role.name` en los claims[34]. Se firma usando una clave secreta segura (definida en el servidor). En UrbanTracker el token por defecto tiene una expiración establecida (por ejemplo 10 horas)[35], tras lo cual ya no será válido.

4. **Respuesta Exitosa:** El backend envía de vuelta la respuesta JSON con el token JWT generado (y opcionalmente la fecha/hora de expiración). Un ejemplo de respuesta JSON podría ser:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expiracion": "2025-10-01T15:30:00Z",
  "userId": 5,
  "role": "DRIVER"
}
```

   El token es una cadena larga codificada en base64 que incluye encabezado, payload y firma. El campo expiración permite a la aplicación saber hasta cuándo es válido.

5. **Almacenamiento Seguro del Token:** La aplicación cliente, al recibir el JWT, lo almacena de forma segura. En las apps móviles se usa **AsyncStorage** (almacenamiento persistente) para guardar el token[36]. En la aplicación web, típicamente se almacenaría en memoria o en *localStorage* de forma segura (evitando exponerlo a JavaScript de terceros para prevenir XSS). Tras login, el token residirá en el cliente y servirá como credencial para llamadas subsecuentes.

6. **Autenticación en solicitudes subsecuentes:** Cada vez que el cliente realiza una petición a un endpoint protegido del backend, incluye el token JWT en la cabecera HTTP `Authorization` usando el esquema *Bearer*. Ejemplo de cabecera:

```
Authorization: Bearer <token_jwt_aquí>
```

El backend, gracias a un filtro de seguridad, intercepta todas las solicitudes entrantes. Si la ruta es pública (por ejemplo, `/api/v1/public/...`), el filtro deja pasar la solicitud sin requerir token[19]. Si la ruta es privada (cualquier otra, p. ej. `/api/v1/vehicle`), el filtro busca la cabecera Authorization y extrae el token JWT[37].

7. **Validación del JWT (Backend):** El filtro JWT valida el token usando la misma clave secreta con la que fue firmado. Verifica que: a) la firma es correcta (no fue modificado el token), b) no esté expirado, y c) los datos concuerdan con un usuario existente. Para esto, se obtiene el username del token[38] y se carga el UserDetails correspondiente (a través de `UserDetailsService`)[39]. Si todo es válido, se considera **autenticado** ese request y se permite el paso al controlador correspondiente. Si falla (token ausente, inválido o expirado), el request es rechazado con error (401).

8. **Sesión y Logout:** Dado que JWT es *stateless*, el backend no mantiene sesión del lado servidor. El "estado" de autenticación reside en el token. Para cerrar sesión (logout), basta con que el cliente elimine el token almacenado. En las apps Expo, esto se implementa borrando el token de AsyncStorage[40]. Después de logout, cualquier petición sin token o con un token eliminado será rebotada. Adicionalmente, en los clientes se implementa un *auto-logout* programado: usando la información de expiración se puede iniciar un temporizador que, tras alcanzarse la hora de expiración, borre el token y fuerce al usuario a autenticarse de nuevo (previniendo uso de tokens caducados).

7.2 Gestión de Tokens en Cliente

Las aplicaciones clientes manejan cuidadosamente el ciclo de vida del JWT:

- **Almacenamiento local:** Como se mencionó, en React Native se emplea AsyncStorage para persistir el token (clave `auth_token`) y datos básicos del usuario autenticado[36]. Esto permite que, si el usuario cierra la app y la vuelve a abrir, la app recuerde su sesión sin tener que loguearse nuevamente (hasta que expire el token). En la app web, se podría usar `localStorage` o cookies seguras (HttpOnly) dependiendo de la estrategia; en nuestro caso, asumimos manejo manual con localStorage ya que es una API REST separada.

- **Inclusión automática en peticiones:** Se procura que cada petición desde el cliente adjunte el token en la cabecera Authorization. Para facilitar esto, típicamente se crea un cliente HTTP central (por ejemplo, una instancia de Axios configurada con un interceptor que añade la cabecera, o una función fetch personalizada). En el contexto de React Native, cada llamada al backend puede leer el token de AsyncStorage y enviarlo. Un pseudoejemplo:

```
const token = await AsyncStorage.getItem('auth_token');
fetch(API_URL + '/vehicle', {
  headers: { 'Authorization': 'Bearer ' + token }
});
```

De esta forma, el token acompaña todas las solicitudes protegidas.

- **Renovación y expiración:** Actualmente, el backend emite tokens con expiración fija (no renovación automática). Por ello, el cliente debe manejar el caso de expiración. La estrategia implementada es calcular la edad del token y, si excede cierto umbral, considerarlo expirado[41]. En la app del conductor, por ejemplo, se obtiene la marca de tiempo con la que se generó el token (en este caso, como es token simulado, extraen un timestamp de la cadena) y se compara con un máximo (24 horas en el ejemplo)[42]. Si excede, se realiza logout automático[43]. En producción, en lugar de este cálculo casero, se podría enviar el campo `expiracion` junto al token y usarlo directamente.

- **Manejo de errores de autenticación:** Si el backend devuelve un error 401 en alguna petición (por token inválido/expirado), las aplicaciones deben capturarlo. La práctica recomendada es interceptar respuestas 401 globalmente: en la app web, redirigir al login; en las apps móviles, limpiar sesión y navegar a la pantalla de inicio de sesión. Esto evita que el usuario continúe en una sesión inválida.

En síntesis, los tokens JWT permiten a UrbanTracker tener una autenticación robusta sin estado de sesión en servidor, adecuada para aplicaciones móviles y SPA. El manejo en cliente se centra en guardar, enviar y borrar el token en el momento adecuado, manteniendo así una **sesión segura**.

7.3 Sistema de Roles y Permisos

Se utiliza un esquema básico de **control de acceso basado en roles (Role-Based Access Control, RBAC)**. Cada usuario en UrbanTracker tiene asociado un rol que determina sus privilegios dentro del sistema. Los roles principales contemplados son: - **ADMIN:** Administrador del sistema (por ejemplo, personal de la compañía de transporte). Tiene acceso total al panel web de administración: gestionar usuarios, crear rutas, asignar vehículos, ver reportes. - **DRIVER (Conductor):** Usuario conductor de bus. Sus permisos se limitan al uso de la app móvil de conductor, la cual básicamente envía su ubicación y le podría mostrar información de su ruta asignada. Un conductor no accede al panel admin ni puede alterar datos sensibles. - **USER (Pasajero):** *Opcional.* Podría existir un rol para usuarios pasajeros registrados en la app móvil de usuario (si se implementa registro/login de pasajeros). Actualmente la app de pasajero puede usarse sin autenticación, por lo que este rol no se usa activamente. En caso de requerirlo, serviría para que pasajeros guarden rutas favoritas, reciban alertas personalizadas, etc.

En la base de datos, los roles se gestionan en una tabla separada (`role`). Cada usuario apunta a un rol (relación muchos-a-uno). Por ejemplo, un registro de usuario tiene un `role_id` que referenciará a la tabla role. La entidad Role contiene al menos un ID y un nombre de rol (como "ADMIN", "DRIVER") y se garantiza la unicidad del nombre[44][45].

Asignación y uso de roles: - Al momento de crear un usuario (por un admin vía API o seeding inicial), se asigna un rol válido. La API de creación de usuarios (`POST /api/v1/public/user`) espera un campo role (por ejemplo un ID numérico del rol) en el DTO[46]. La lógica de UserService comprueba que el rol exista; si no, lanza error[47]. - En el token JWT, como se mencionó, se incluye el rol del usuario en los claims[34]. Por ejemplo, un JWT puede tener `"role": "ADMIN"`. Esto permite que el cliente o incluso el backend usen esa info sin necesidad de otra consulta. - **Restricción en backend:** Actualmente, **no se ha implementado una restricción fina por anotaciones de seguridad** (ej. `@PreAuthorize("hasRole('ADMIN')")`) en los controladores. Sin embargo, dado que solo administradores tendrían el token para acceder a ciertas rutas, se asume que la separación de apps previene que roles equivocados llamen endpoints no destinados a ellos. En futuras versiones, se podría agregar filtros adicionales: por ejemplo, asegurar que los endpoints de administración solo los acceda un usuario con rol ADMIN (se puede hacer manual en el controller obteniendo el usuario autenticado y chequeando su rol, o mediante anotaciones de Spring Security). - **Uso en el frontend:** En la aplicación web, se podría implementar lógica de UI condicional según el rol. Por ejemplo, si un usuario no es admin, ocultar ciertas secciones. En nuestro caso, la app web es solo para admins, por lo que asume ADMIN siempre. En las apps móviles, el rol DRIVER se refleja en que usan la app de conductor separada, y un pasajero usaría la otra app.

Como buena práctica, en el código móvil se pueden incluir funciones utilitarias como:

```
function hasRole(user, requiredRole) {
  return user?.role === requiredRole;
}
```

y usarlo para condicionar opciones de la interfaz (esto es análogo a lo que se haría con roles en front-end web). En pseudo-código:

```
{ hasRole(currentUser, 'ADMIN') && <AdminMenu /> }
```

Así solo admins ven ciertas opciones.

7.4 Mejores Prácticas de Seguridad de Datos

UrbanTracker adopta varias prácticas para proteger los datos de los usuarios y la integridad del sistema: - **Comunicación Segura (HTTPS):** En entornos de producción, todo el tráfico entre clientes y servidor debe cifrarse usando HTTPS. Esto previene

ataques de tipo *man-in-the-middle* y resguarda tanto las credenciales como los tokens JWT en tránsito. (En desarrollo local se puede usar HTTP simple, pero nunca en producción). - **Contraseñas Hasheadas:** Las contraseñas de usuario nunca deben almacenarse en texto plano. El backend está preparado con un `PasswordEncoder` de Spring (BCrypt)[48]; aunque en la implementación actual la función de guardado de usuario no lo utiliza explícitamente (debería integrarse), se asume que las contraseñas se guardarán cifradas. BCrypt agrega *salt* y es resistente a ataques de fuerza bruta. **Se recomienda encarecidamente** actualizar la lógica de `UserService.save` para aplicar `passwordEncoder.encode(contraseña)` antes de guardar el usuario. - **Token JWT seguro:** La clave secreta usada para firmar el JWT es robusta (una cadena aleatoria Base64 de 32 bytes)[49]. Esta clave se debe mantener privada; idealmente almacenada en una variable de entorno del servidor y no hardcodeada. Un token JWT por sí mismo no contiene datos sensibles aparte de identificadores, pero si un atacante lo obtiene podría suplantar la sesión del usuario hasta que expire. Por ello, es importante: - En el cliente, proteger el token (no exponerlo en logs, no guardarlo en lugares inseguros). - En el backend, definir un tiempo de expiración razonable (10 horas en dev; en producción quizá 1-2 horas) para limitar la ventana de uso si es robado. - Opcionalmente implementar *refresh tokens* o lista de revocación si se requiere invalidar tokens antes de su expiración (no implementado actualmente). - **Políticas CORS y CSRF:** El backend al ser API REST para clientes específicos (web y móviles) habilita CORS para las direcciones necesarias (p. ej., la app web corriendo en `http://localhost:3000`). Se configura Spring Security para permitir esas solicitudes. Al ser JWT, no se usa protección CSRF de formulario. La aplicación web debe emitir las peticiones AJAX con el header Authorization adecuado. - **Validación de Entradas:** En todos los endpoints de escritura, se valida la información entrante. La lógica en los servicios comprueba campos requeridos, formatos (por ejemplo, que el username no tenga caracteres inválidos, que el email de contacto de Company tenga formato correcto[50], etc.), y longitudes máximas. Esto evita SQL injections (JPA Queries parametrizadas) y cualquier entrada maliciosa que pueda causar comportamientos inesperados. - **Manejo de Errores Seguro:** El backend encapsula las respuestas en objetos ResponseDTO indicando el resultado. En caso de error, se provee un mensaje genérico o específico, pero sin exponer trazas internas. Por ejemplo, si falla el login, se devuelve 401 con “Credenciales inválidas”, en lugar de una excepción detallada. Igualmente, se podría configurar un control global de excepciones para loguear internamente los errores pero responder con mensajes seguros hacia fuera. - **Limpieza de Datos Sensibles:** Cuando un usuario cierra sesión, las apps eliminan toda la información sensible almacenada (token, datos de usuario)[40]. También, si un conductor deja de usar la app, debe terminar el envío de ubicación. - **Permisos Móviles:** Las aplicaciones solicitan únicamente los permisos necesarios. En el caso del conductor, se piden permisos de Localización Fine/Coarse y en segundo plano[18]. Estos permisos están justificados por la funcionalidad. No se solicitan permisos innecesarios (como contactos, almacenamiento) que pudieran poner en riesgo privacidad. En el app.json se documenta el uso de la ubicación para

transparencia con el usuario. - **Protección ante Providers de Ubicación:** Un caso específico: se detectó un error de *"No valid location provider" en Android. Se actualizó la configuración para incluir permisos de ubicación en AndroidManifest y guiar al usuario a activar el GPS[22][51]. Esto mejora la disponibilidad del dato de localización, previniendo escenarios donde la app quede sin capacidad de rastrear debido a ajustes del dispositivo.

En conjunto, estas medidas aseguran que UrbanTracker maneje adecuadamente la seguridad a nivel de autenticación, autorización y protección de datos, brindando confianza tanto a los administradores (que sus datos están seguros) como a los usuarios finales. Siempre es recomendable someter la aplicación a pruebas de penetración y revisar periódicamente las dependencias por actualizaciones de seguridad (por ejemplo, nuevas versiones de Spring Boot corrigiendo vulnerabilidades).

8. Especificación de API REST

A continuación, se describen los principales endpoints RESTful que expone el backend de UrbanTracker, incluyendo sus rutas, métodos HTTP, formatos de request/response y códigos de estado esperados. La API sigue el estilo REST, con rutas organizadas por recurso y usando los verbos HTTP estándar (GET, POST, DELETE, etc.). Todas las respuestas se devuelven en formato **JSON**.

8.1 URL Base

En desarrollo, el servidor API está disponible típicamente en:

```
http://localhost:8080/api/v1/
```

Esta es la URL base sobre la cual se montan los distintos endpoints (añadiendo el segmento correspondiente al recurso). En producción, esta URL cambiará según el dominio o IP del servidor donde se despliegue. Por ejemplo, podría ser `https://miempresa.com/urbantracker/api/v1/`. UrbanTracker incorpora Swagger/OpenAPI para documentación interactiva; al levantar el backend, se puede acceder a la interfaz Swagger UI en `http://localhost:8080/swagger-ui/index.html` para explorar y probar los endpoints.

8.2 Formato de Respuestas

Todas las respuestas de la API se devuelven en JSON, usando convenciones consistentes para indicar éxito o error. En general: - **Respuesta exitosa (códigos 2xx):** Se devuelve un objeto JSON conteniendo los datos solicitados o un mensaje de éxito. Puede incluir un campo `status` con el código HTTP o descripción, y campos como `data` o `message`. En las consultas (GET), típicamente se devuelve directamente la lista o el objeto solicitado. En operaciones de creación/actualización/borrado (POST/DELETE), se suele envolver en un objeto de tipo ResponseDTO con un

mensaje. *Ejemplo:*

```
{
  "status": "200 OK",
  "data": [
     { "id": 1, "username": "admin", "role": 1 },
     { "id": 2, "username": "driver1", "role": 2 }
  ]
}
```

En este ejemplo, data contiene una lista de usuarios (cada uno con su id, nombre de usuario y role id).

- **Respuesta de error (códigos 4xx/5xx):** Se devuelve un objeto indicando el error ocurrido. Puede incluir `status` (código o nombre), un campo `error` o `message` con la descripción, y opcionalmente detalles adicionales. *Ejemplo:*

  ```
  {
    "status": "404 NOT_FOUND",
    "error": "El recurso solicitado no existe"
  }
  ```

  O para errores de validación:

  ```
  {
    "status": "400 BAD_REQUEST",
    "error": "Datos de entrada inválidos",
    "details": [
        "El nombre no puede estar vacío",
        "El email no tiene formato válido"
    ]
  }
  ```

  En este caso se proporciona un array `details` con mensajes específicos por campo.

El API utiliza códigos HTTP significativos: 200 OK en lecturas exitosas, 201 Created cuando se crea un nuevo recurso, 400 Bad Request para datos incorrectos, 401 Unauthorized para falta de autenticación, 403 Forbidden para falta de permisos, 404 Not Found si no existe el recurso, 409 Conflict para violaciones de unicidad, y 500 Internal Server Error para excepciones no controladas.

8.3 Autenticación (Login)

`POST /api/v1/auth/login` – Autentica a un usuario y devuelve un token JWT.
- **Descripción:** Verifica las credenciales enviadas (usuario y contraseña). Si son válidas, genera un token JWT para sesiones subsecuentes.

- **Request:**

```
{
  "username": "conductor1",
  "password": "secreto123"
}
```

- **Response (200 OK):**

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expiracion": "2025-10-01T15:30:00Z",
  "userId": 7,
  "role": "DRIVER"
}
```

El campo `token` contiene el JWT que el cliente debe guardar. `expiracion` indica la validez (opcional). Se pueden incluir datos básicos del usuario para que el cliente no tenga que pedirlos inmediatamente después.
- **Errores:**
- 401 Unauthorized – Credenciales inválidas (usuario o contraseña incorrectos).
Ejemplo: `{ "status": "401 UNAUTHORIZED", "error": "Credenciales inválidas" }`
- 400 Bad Request – Formato de datos incorrecto (por ejemplo, falta el campo username).

8.4 Endpoints de Usuario (`/user`)
Operaciones para gestionar usuarios del sistema.

- `GET /api/v1/public/user` – **Listar Usuarios.** Retorna la lista de todos los usuarios registrados[7].
- **Requiere Auth:** No (endpoint público, aunque en un entorno real esto debería ser privado para ADMIN).
- **Response (200):** Array JSON de usuarios. Cada usuario incluye id, username, role (posiblemente como id o nombre). Ejemplo:

  ```
  [
    { "id": 1, "username": "admin", "role": 1 },
    { "id": 2, "username": "driver1", "role": 2 }
  ]
  ```

  Aquí "role": 1 podría corresponder a ADMIN y 2 a DRIVER.
- `POST /api/v1/public/user` – **Crear nuevo Usuario.** Registra un nuevo usuario con los datos proporcionados[52].

- **Requiere Auth:** No (público). En un caso de uso real, esto podría emplearse para registro abierto de conductores/pasajeros o bien debería ser privado para que solo admin cree usuarios.
- **Request:** JSON con datos del usuario. Ejemplo:

```
{
  "userName": "nuevoConductor",
  "password": "abc12345",
  "role": 2,
  "company": 5
}
```

  Donde `role` es el ID del rol (2 = DRIVER) y `company` podría ser el ID de la compañía transportista a la que pertenece (si aplica).
- **Response (200 OK):** Devuelve un objeto de respuesta estándar indicando éxito. Por ejemplo:

```
{
  "status": "200 OK",
  "message": "El usuario se guardó correctamente."
}
```

  (Usa ResponseDTO internamente[53]).
- **Errores:** 400 si fallan validaciones (e.g. nombre de usuario vacío o ya existe, contraseña muy corta), 409 Conflict si el usuario ya existía[14]. Los mensajes de error detallan la causa, p.ej.: *"Ya existe un usuario con ese nombre."*.
- `DELETE /api/v1/public/user/{id}` - **Eliminar Usuario.** Borra al usuario con el ID especificado[54].
- **Requiere Auth:** No (público). En la práctica, debería requerir autenticación de admin para usarse.
- **Response (200):**

```
{ "status": "200 OK", "message": "user eliminado correctamente" }
```

  (El mensaje original tiene una minúscula "user", pero conceptualmente indica éxito)[55].
- **Errores:** 404 Not Found si el ID no existe.

Nota: Actualmente no hay un endpoint dedicado para obtener detalles de un usuario por ID (`GET /api/v1/user/{id}`) ni para actualizar un usuario (`PUT /user/{id}`) en la rama dev. La creación (POST) se utiliza tanto para nuevos como para actualizar existentes si se envía un `id` en el JSON (la lógica en UserService distingue id nulo/nuevo vs existente para actualizar)[56]. Sin embargo, no exponer un PUT/PATCH es algo a considerar en el futuro.

8.5 Endpoints de Roles (`/role`)

Gestión de los roles de usuario. Estos endpoints requieren autenticación (y lógicamente, solo un ADMIN debería usarlos).

- `GET /api/v1/role` – **Listar Roles.** Devuelve todos los roles definidos (ej. ADMIN, DRIVER)[57].
- **Auth:** Sí, requiere token válido (admin).
- **Response:** Lista JSON de roles. Ejemplo:

```
[
  { "id": 1, "name": "ADMIN" },
  { "id": 2, "name": "DRIVER" }
]
```

- `POST /api/v1/role` – **Crear Rol.** Crea un nuevo rol.
- **Auth:** Sí (admin).
- **Request:** JSON con el nombre del nuevo rol: `{ "name": "SUPERVISOR" }`.
- **Response:** Mensaje de éxito: `{ "status": "200 OK", "message": "El rol se guardó correctamente." }`[58].
- **Errores:** 400 si el nombre es inválido (vacío, muy largo, caracteres no permitidos)[59][45]; 409 Conflict si ya existe un rol con ese nombre.
- `DELETE /api/v1/role/{id}` – **Eliminar Rol.** Borra un rol por ID (solo si no tiene usuarios asociados típicamente).
- **Auth:** Sí (admin).
- **Response:** `{ "status": "200 OK", "message": "Rol eliminado correctamente" }`[60].
- **Errores:** 404 si el rol no existe, 400 si el rol no se puede borrar por alguna razón de integridad (no manejado explícitamente, pero podría haber restricción referencial a usuarios).

8.6 Endpoints de Vehículos (`/vehicle`)

Administración de los buses o vehículos de transporte.

- `GET /api/v1/vehicle` – **Listar Vehículos.** Retorna todos los vehículos registrados[61].
- **Auth:** Sí (ADMIN token).
- **Response:** Array JSON de vehículos. Cada objeto vehículo podría tener: id, licencePlate, brand, model, company, status, user (conductor asignado). Ejemplo simplificado:

```
[
  {
    "id": 10,
    "licencePlate": "ABC123",
```

```
    "brand": "Mercedes",
    "model": "Sprinter",
    "company": 5,
    "status": "ACTIVE",
    "user": 2
  },
  { ... }
]
```

Aquí `company` es el ID de la compañía dueña, `user` es el ID del conductor asignado, `status` es el estado del vehículo (posiblemente un string).

- `POST /api/v1/vehicle` – **Crear/Actualizar Vehículo.** *(En la rama dev, este endpoint no estaba implementado aún, pero lógicamente existirá similar al de user/role.)*
- **Auth:** Sí (admin).
- **Request:** JSON con datos del vehículo, p.ej.:

```
{
  "licencePlate": "XYZ789",
  "brand": "Volvo",
  "model": "7700",
  "company": 5,
  "status": 1,
  "user": 3
}
```

Donde status quizás sea un ID referenciando la entidad VehicleStatus (1=ACTIVE, 2=INACTIVE), o un booleano.

- **Response:** Mensaje de éxito con status 200 o 201.
- **Errores:** 400 si faltan datos o son inválidos (ej: placa vacía), 409 si la placa ya existe (suponiendo que es única).
- `GET /api/v1/vehicle/{id}` – **Obtener Vehículo (Detalle).** *(A implementar)* Retornaría los datos de un vehículo específico.
- **Auth:** Sí (admin o quizás conductor si solo puede ver su vehículo).
- **Response:** Objeto del vehículo (mismo formato que en lista).
- **Errores:** 404 si el ID no existe.
- `DELETE /api/v1/vehicle/{id}` – **Eliminar Vehículo.** Borra un vehículo.
- **Auth:** Sí (admin).
- **Response:** `{ "status": "200 OK", "message": "Vehículo eliminado correctamente." }`.
- **Errores:** 404 si no existe.

8.7 Endpoints de Rutas (`/route`)

Administración de las rutas de transporte público.

- `GET /api/v1/route` – **Listar Rutas.** Devuelve todas las rutas existentes[62].
- **Auth:** Sí (admin).
- **Response:** Array JSON de rutas. Cada ruta podría incluir: id, nombre o número de ruta, origen, destino, compañía operadora, etc. Ejemplo:

```
[
  { "id": 101, "name": "Ruta A - Centro", "company": 5 },
  { "id": 102, "name": "Ruta B - Circular", "company": 5 }
]
```

(Campos adicionales como una descripción, horario, etc., según modelado).

- `POST /api/v1/route` – **Crear/Actualizar Ruta.** Crea una nueva ruta o actualiza una existente.
- **Auth:** Sí (admin).
- **Request:**

```
{
  "name": "Ruta C - Nuevo Barrio",
  "company": 5,
  "points": [ { "lat": 1.234, "lng": -77.001 }, ... ]
}
```

(Podría contener incluso la lista de puntos de ruta que componen el recorrido, o eso se maneja separadamente en route-point).

- **Response:** 200 OK con mensaje de éxito.
- **Errores:** 400 si datos inválidos, 409 si una ruta con ese nombre ya existe para la misma compañía.
- `GET /api/v1/route/{id}` – **Obtener Ruta Detallada.** Retorna datos de una ruta específica, posiblemente incluyendo sus puntos.
- **Auth:** Sí.
- **Response:**

```
{
  "id": 101,
  "name": "Ruta A - Centro",
  "company": { "id": 5, "name": "Empresa XYZ" },
  "points": [
     { "id": 1001, "lat": 1.234, "lng": -77.001, "order": 1 },
     ...
  ]
}
```

Este formato incluiría objetos anidados como la compañía y un arreglo de puntos de ruta ordenados.

- `DELETE /api/v1/route/{id}` – **Eliminar Ruta.**
- **Auth:** Sí.
- **Response:** Confirmación de borrado.
- **Errores:** 404 si no existe, 400 si no se puede borrar (por ejemplo, si tiene viajes históricos asociados; en tal caso habría que decidir cómo manejarlos).

8.8 Endpoints de Puntos de Ruta (`/route-point`)
Corresponden a los puntos geográficos (paradas o coordenadas intermedias) que conforman cada ruta.

- `GET /api/v1/route-point` – **Listar Puntos de Ruta.** Devuelve todos los puntos de todas las rutas (posiblemente no muy útil listar todos sin filtro, pero así está en dev)[63].
- **Auth:** Sí.
- **Response:** Array JSON con todos los puntos. Probablemente cada punto incluye id, latitud, longitud, orden y referencia a la ruta:

```
[
  { "id": 1001, "routeId": 101, "lat": 1.234, "lng": -77.001,
"order": 1 },
  { "id": 1002, "routeId": 101, "lat": 1.235, "lng": -77.010,
"order": 2 },
  ...
]
```

  En general, se accedería a los puntos de una ruta específica mediante el endpoint de detalle de ruta o un filtro (no implementado explícitamente, pero podría ser GET `/route/{id}/points`).
- `POST /api/v1/route-point` – **Crear Punto de Ruta.** Agrega un nuevo punto a una ruta.
- **Auth:** Sí.
- **Request:**

```
{
  "route": 101,
  "lat": 1.250,
  "lng": -77.050,
  "order": 15
}
```

  Insertaría un nuevo punto en la ruta con el orden indicado.
- **Response:** 200 OK con datos del punto creado (ej. con su id asignado).
- **Errores:** 400 si datos inválidos, p.ej. lat/lng faltantes.
- `DELETE /api/v1/route-point/{id}` – **Eliminar Punto de Ruta.**

- **Auth:** Sí.
- **Response:** Mensaje de éxito.
- **Errores:** 404 si no existe.

8.G Endpoints de Viajes (`/journey`)
Representan instancias de recorrido efectuadas. Un "viaje" podría generarse cada vez que un vehículo recorre una ruta en un horario específico.

- `GET /api/v1/journey` – **Listar Viajes.** Retorna todos los viajes registrados (por ejemplo, históricos o en curso)[64].
- **Auth:** Sí (admin).
- **Response:** Lista de viajes. Un viaje puede contener: id, ruta, vehículo, hora de inicio, hora de fin, estado (en curso, finalizado). Ejemplo:

```
[
  { "id": 501, "route": 101, "vehicle": 10, "startTime": "2025-10-01T08:00:00Z", "endTime": "2025-10-01T09:30:00Z" },
  { "id": 502, "route": 102, "vehicle": 12, "startTime": "2025-10-01T08:15:00Z", "endTime": null }
]
```

  Aquí el segundo objeto representa un viaje en curso (endTime null).
- `POST /api/v1/journey` – **Registrar nuevo Viaje.** Inicia un nuevo viaje (por ejemplo, cuando un bus comienza su ruta).
- **Auth:** Sí.
- **Request:**

```
{
  "route": 101,
  "vehicle": 10,
  "startTime": "2025-10-01T14:00:00Z"
}
```

  (El startTime podría ser opcional si se toma el actual).
- **Response:** Datos del viaje creado, incluyendo su id.
- `PUT /api/v1/journey/{id}/finish` – **Finalizar Viaje.** Marca la hora de finalización de un viaje en curso.
- **Auth:** Sí.
- **Request:** (no body, solo path)
- **Response:** Viaje actualizado con endTime.
- `GET /api/v1/journey/{id}` – **Detalle de Viaje.** Podría incluir todos los puntos o eventos del viaje (por ejemplo, registro de posiciones, tiempos entre paradas, etc., si se almacenan). *(No implementado explícitamente en dev.)*

Endpoints en Tiempo Real:
Además de los endpoints REST tradicionales arriba descritos, UrbanTracker maneja actualizaciones de ubicación en tiempo real, que no siguen el modelo request/response sino una comunicación continua: - Si se utiliza **WebSockets/Socket.IO**, habría endpoints como `ws://<server>/socket.io` donde el cliente se conecta. Una vez conectado, se pueden emitir eventos como "locationUpdate" desde el servidor con payloads de posiciones, y los clientes (web/pasajero) los reciben y manejan. El detalle de los eventos y mensajes se documentaría aparte, pero conceptualmente: cada actualización contendría un identificador de vehículo y coordenadas nuevas, p.ej. `{ vehicleId: 10, lat: ..., lng: ..., timestamp: ... }`. La suscripción a qué vehículos/rutas seguir se podría basar en join a "rooms" de Socket.IO (p. ej., un pasajero se une a la "sala" de la ruta 101 para recibir solo de esa). - Si se utiliza **MQTT**, los topics podrían ser jerárquicos, por ejemplo: `urbantracker/route/101` donde se publican mensajes de posición de cualquier bus en ruta 101. Los clientes se suscriben a los topics de interés. Los mensajes MQTT podrían tener formato JSON similar al mencionado. En este esquema, los endpoints de la API REST tradicional servirían para configuraciones y datos estáticos, mientras la ubicación en vivo pasa por el broker.

Para fines de este documento, enfatizamos los endpoints HTTP REST, ya que los de tiempo real son de otra naturaleza. Sin embargo, es importante saber que la aplicación web y las apps móviles usan esos canales en paralelo: obtienen datos iniciales vía REST (por ejemplo, lista de rutas, estado actual) y luego reciben actualizaciones incrementales vía Socket/MQTT.

Resumen: La API de UrbanTracker ofrece las funcionalidades CRUD básicas para las entidades principales (usuarios, roles, vehículos, rutas, puntos, compañías) y endpoints específicos para flujos (login, etc.). La versión actual (v1) es funcional pero básica; futuras extensiones podrían incluir filtrado/paginación en listados, búsqueda por atributos (ej: vehículos por compañía), y endpoints más refinados (ej: obtener las rutas asignadas a un conductor específico, etc.). La documentación de Swagger disponible en la app facilita a los desarrolladores explorar cada endpoint en detalle y probarlos durante la integración.

G. Modelos de Datos

Esta sección describe los principales modelos de datos (entidades) utilizados en el backend de UrbanTracker. Cada modelo corresponde a una tabla en la base de datos PostgreSQL y está implementado como una clase Java con anotaciones JPA (`@Entity`). Se incluyen sus campos más importantes, tipos de datos y relaciones con otras entidades.

G.1 Usuario (User)

Representa a un usuario del sistema (puede ser un administrador, conductor, etc.). Implementa `UserDetails` de Spring Security para integrarse con la autenticación.

Campos principales de `User`[65][66]: - **id:** Integer, clave primaria autogenerada (IDENTITY). Identificador único de usuario.
- **username:** String, nombre de usuario único (login). Longitud hasta 50. *Ej:* `"admin"`, `"chofer123"`.
- **password:** String, contraseña del usuario en formato cifrado (BCrypt). Longitud hasta 50. *Ej:* `"$2a$10$Xy... (hash)"`.
- **role:** Role, referencia (ManyToOne) al rol del usuario. Un usuario pertenece a un rol (admin, driver, etc.)[67]. Almacena internamente el `role_id`.
- **vehicles:** List<Vehicle>, relación OneToMany inversa hacia Vehicle. Indica los vehículos asignados a este usuario (aplica típicamente a conductores que tienen vehículos asignados)[68]. Es marcada `@JsonIgnore` para evitar recursión infinita en JSON.

Otras propiedades (heredadas de UserDetails): - **accountNonExpired, accountNonLocked, credentialsNonExpired, enabled:** booleanos de control de cuenta. Actualmente `isEnabled()` devuelve una constante (en la implementación actual toma un campo estático no mostrado, pero se asume true si activo)[69][70]. Estos podrían usarse en un futuro para deshabilitar cuentas.

Relaciones: Muchos usuarios pueden tener el mismo rol (ManyToOne hacia Role). Un usuario puede tener múltiples vehículos (OneToMany hacia Vehicle). Si se añade relación con Company, podría ser ManyToOne (un usuario asociado a una compañía, especialmente conductores perteneciendo a una empresa de transporte). Esto último no está explícito en la entidad User actual.

G.2 Rol (Role)

Define un rol o perfil de permisos en el sistema.

Campos de `Role` (inferido de uso, ya que la clase Role no se mostró directamente en búsquedas, pero por IRole y contexto): - **id:** Integer, clave primaria.
- **name:** String, nombre del rol (único)[71], por ejemplo `"ADMIN"`, `"DRIVER"`. Longitud hasta 50.

Posibles campos adicionales: - **users:** List<User>, lista de usuarios que tienen este rol (mapeo OneToMany inverso). Si existe, se anotaría con `@OneToMany(mappedBy="role")`. Probablemente marcado con `@JsonIgnore` para evitar recursión.

La tabla `role` contiene registros fijos (p.ej., 1=ADMIN, 2=DRIVER). El sistema no necesita muchos campos en Role más allá del nombre para comparar permisos.

G.3 Compañía (Company)

Representa una empresa operadora de transporte (p. ej., una compañía de buses). Se utiliza para agrupar vehículos y posiblemente usuarios administradores por empresa.

Campos de `Company`[72][73]: - **id:** Integer, clave primaria.
- **name:** String, nombre legal o identificador de la compañía. *Ej:* `"Transportes Urbanos S.A."`.
- **nit:** String, número de identificación tributaria (NIT) único[74]. Se usa en Colombia como identificador único de empresas. Solo dígitos.
- **address:** String, dirección física principal de la compañía[74].
- **contactPhone:** String, teléfono de contacto (puede incluir +, -, espacios)[75]. Longitud 10 en BD (probablemente para número local).
- **contactEmail:** String, correo electrónico de contacto[76].
- **active:** boolean, indica si la compañía está activa o deshabilitada[77] (podría usarse para filtrar en listados si fuese necesario).
- **createAt:** LocalDateTime, timestamp de creación (asignado automáticamente en @PrePersist)[78][79].
- **updateAt:** LocalDateTime, timestamp de última actualización (actualizado en @PreUpdate)[78][80].

Relaciones: - **vehicles:** List<Vehicle>, relación OneToMany hacia Vehicle (una compañía tiene muchos vehículos)[81]. Los vehículos conocen su `company` dueña. Marcado con JsonIgnore en Company para evitar recursión.

La entidad Company ayuda a segmentar la información cuando hubiera múltiples empresas usando el sistema, o simplemente para registrar datos de la única empresa operadora.

G.4 Vehículo (Vehicle)

Representa un vehículo de transporte (bus). Contiene la información del bus y sus asignaciones.

Campos de `Vehicle`[82][83]: - **id:** Integer, clave primaria.
- **licencePlate:** String, placa del vehículo (única). Ej: `"ABC123"`[84].
- **brand:** String, marca del vehículo. Ej: `"Mercedes-Benz"`[85].
- **model:** String, modelo o línea del vehículo. Ej: `"Sprinter 415"`[85].
- **status:** VehicleStatus, referencia ManyToOne al estado actual del vehículo[86]. Puede ser una entidad enumerada con valores como ACTIVO/INACTIVO, o EN_SERVICIO/FUERA_DE_SERVICIO, etc. Este campo indica si el vehículo está operativo.
- **company:** Company, referencia ManyToOne a la compañía dueña del vehículo[82]. Es obligatorio (cada vehículo pertenece a una empresa).
- **user:** User, referencia ManyToOne al usuario asignado al vehículo[87]. En contexto, se refiere al conductor asignado actualmente a ese vehículo. Debe tener role = DRIVER. Este campo es obligatorio en la BD (nullable=false), lo que sugiere que siempre debe haber un conductor asignado; sin embargo, en la vida real podría no ser

siempre el caso, quizá se asigne un "placeholder" o se actualice a medida que cambian turnos.

Relaciones inversas: - En User, el campo vehicles mapea la relación desde el lado opuesto, permitiendo acceder a todos los vehículos de un conductor (OneToMany). - En Company, el campo vehicles mapea todos los vehículos de la empresa (OneToMany).

G.5 Estado de Vehículo (VehicleStatus)

Indica el estado operativo de un vehículo.

Campos de `VehicleStatus`[44]: - **id:** Integer, clave primaria.
- **name:** String, nombre del estado (único). Ej: `"ACTIVE"`, `"INACTIVE"`[88]. En la BD posiblemente hay registros predefinidos. - *(No se muestran relaciones, pero asumo)* **vehicles:** List<Vehicle> (OneToMany inverso) de vehículos que tienen ese estado.

En la implementación actual, parece que tenían tanto una entidad `VehicleStatus` en modelo (para BD)[44], como un enum `VehicleStatus` en otro paquete[89]. Es probable que acabaran usando la entidad para poder almacenar los estados en tabla y referenciarlos con clave foránea. Los valores típicos serían: - 1: ACTIVE (vehículo operativo)
- 2: INACTIVE (no operativo, mantenimiento)
Se podría extender a más estados (EN_RUTA, etc.), pero por simplicidad se maneja activo/inactivo.

G.6 Ruta (Route)

Define una ruta de bus (recorrido fijo con paradas específicas).

Campos de `Route` (deducido, ya que no vimos la clase en texto, pero por contexto): - **id:** Integer, clave primaria.
- **name:** String, nombre o código de la ruta. Ej: `"Ruta 5 - Centro a Norte"`. Podría ser un número o identificador corto también.
- **description:** String, descripción de la ruta (opcional, podría incluir información de recorrido).
- **company:** Company, ManyToOne a la empresa que opera la ruta (si aplica que cada ruta pertenece a una empresa determinada).
- **active:** boolean, para indicar si la ruta está activa.
- **points:** List<RoutePoint>, OneToMany con los puntos geográficos que conforman el recorrido.
- **vehicles:** List<Vehicle> (opcional), vehículos asignados a esta ruta actualmente. Esto podría no ser un campo persistido fijo, sino calculado basado en qué journeys están en curso. Probablemente no hay esta relación en la entidad Route.

Ruta sirve principalmente para agrupar los puntos y para asignar viajes.

G.7 Punto de Ruta (RoutePoint)

Un punto de ruta representa una parada o una coordenada en el camino de una ruta.

Campos de `RoutePoint` (deducidos): - **id:** Integer, clave primaria.
- **lat:** Double, latitud geográfica del punto.
- **lng:** Double, longitud geográfica del punto.
- **order:** Integer, orden secuencial del punto en la ruta (1, 2, 3, ...).
- **route:** Route, ManyToOne referencia a la ruta a la que pertenece el punto. Este campo define la relación de pertenencia.

Puede haber campos adicionales como nombre de la parada (`name`, si cada punto representa una parada con nombre). Pero al menos lat/lng/order son esperables.

La tabla route_point usualmente tendría una restricción única compuesta (route_id, order) para que no se repita el orden en la misma ruta, o (route_id, lat, lng) para no duplicar coordenadas exactas.

G.8 Viaje (Journey)

Representa un recorrido efectuado por un vehículo en una ruta, en un tiempo determinado (por ejemplo, el viaje de las 8:00 AM en la Ruta 5 con el bus X).

Campos de `Journey` (deducidos): - **id:** Integer, clave primaria.
- **route:** Route, ManyToOne a la ruta que se recorrió.
- **vehicle:** Vehicle, ManyToOne al vehículo que realizó el viaje.
- **startTime:** Timestamp (DateTime), hora de inicio del viaje.
- **endTime:** Timestamp, hora de finalización del viaje (null si en curso).
- **status:** Enum o indicador del estado del viaje (ej: `ONGOING`, `COMPLETED`). Podría derivarse de endTime null o no, por lo que quizás no haya un campo separado de status.
- **pointsCovered:** (Opcional) Podría almacenarse una traza de puntos realmente cubiertos o eventos durante el viaje, pero eso usualmente se deriva de otra fuente (por ejemplo, registros de telemetría). Probablemente no se guarda en esta entidad, sino que se podrían reconstruir a partir de logs de ubicación.
- **driver:** User (conductor), ManyToOne al usuario que condujo (en realidad esto es redundante si vehicle->user ya enlaza al conductor, pero en caso de reasignaciones podría interesar guardar quién conducía en ese viaje independientemente de la asignación actual del vehículo).

El Journey permite tener un historial. Cada viaje pertenece a una ruta y un vehículo (y por ende a una compañía). Esto sirve para cálculos de estadísticas, monitoreo de puntualidad, etc.

En la implementación actual, la JourneyService solo listaba todos los journeys sin criterios[90], lo cual sugiere que Journey es una entidad simple sin mucha lógica asociada todavía.

Relaciones entre modelos (resumen):
- User --< Role (muchos usuarios por rol).
- Company --< Vehicle (compañía tiene múltiples vehículos).
- Company --< Route (posiblemente, empresa opera varias rutas).

- Role --< User (rol tiene múltiples usuarios).
- User --< Vehicle (un conductor puede estar asociado a varios vehículos, aunque en la práctica conduce de a uno; esto podría representar historial).
- VehicleStatus --< Vehicle (estado asignado a muchos vehículos).
- Route --< RoutePoint (una ruta contiene muchos puntos secuenciales).
- Route --< Journey (una ruta tiene múltiples viajes históricos).
- Vehicle --< Journey (un vehículo realiza múltiples viajes en el tiempo).
- User (Driver) --< Journey (un conductor realiza múltiples viajes).

A continuación, se presenta un **diagrama simplificado** de las entidades y sus relaciones (en formato texto):

```
User *--1 Role
User 1--* Vehicle
Company 1--* Vehicle
Company 1--* Route
Vehicle *--1 Company
Vehicle *--1 User (Driver)
Vehicle *--1 VehicleStatus
Route 1--* RoutePoint
Route 1--* Journey
Vehicle 1--* Journey
User (Driver) 1--* Journey
```

En el diagrama `A 1--* B` indica "A tiene uno a muchos B". Por ejemplo, Company 1--* Vehicle significa "Una Company tiene muchos Vehicles".

Notas sobre integridad referencial:
- Al eliminar una Company, seguramente se eliminan en cascada sus vehículos (o primero habría que eliminar vehículos manualmente porque tienen FK a company). En la implementación actual, CompanyService.delete simplemente borra por ID tras verificar existencia[91]. Dado que se definió `cascade = CascadeType.ALL` en Company.vehicles[81], es probable que borrar la compañía borre automáticamente sus vehículos (cascade all). - Similar para User -> Vehicle (en User.vehicles se definió cascade = ALL[68]). Esto es un poco peligroso, ya que borrar un usuario conductor podría borrar vehículos, lo cual tal vez no es deseado a menos que la intención sea que cada conductor "propietario" de sus vehículos. Es más lógico que eliminar un usuario conductor simplemente deje los vehículos sin asignar. Este detalle de cascade quizá sea revisado en versiones futuras. - Por simplicidad en dev, se usó cascade en varios lados, pero en un entorno real se tendría cuidado para no borrar cascadas inadvertidamente. - Unique constraints: se esperarían constraints únicas en campos como User.username (sí hay unique = true en username[92]), Company.nit (unique = true[74]), Company.contactEmail (unique = true[76]), Vehicle.licencePlate (unique true, no especificado en código visto pero suele ser). Role.name (unique true, está en IRole check y presumably anotado en model). VehicleStatus.name (unique true está anotado[71]).

Este esquema de datos permite cubrir las funcionalidades: usuarios con roles gestionando flotas de vehículos (cada uno con estado y asignación a conductores), recorriendo rutas definidas por puntos. Los datos de Journey complementan con información dinámica.

10. Flujo3 de Negocio

En esta sección se describen algunos de los flujos de negocio principales de UrbanTracker, mediante pasos secuenciales y (cuando aplica) diagramas lógicos. Los flujos explican **cómo interactúan los componentes** del sistema para lograr casos de uso clave, como la autenticación de un usuario, el seguimiento en tiempo real de buses y la gestión administrativa de entidades.

10.1 Flujo de Inicio de Sesión (Login JWT)

Este flujo ocurre cuando un administrador accede al panel web o un conductor abre su app móvil e ingresa sus credenciales. El proceso, integrado con JWT, es el siguiente:

1. **Ingreso de Credenciales:** El usuario ingresa su nombre de usuario y contraseña en la interfaz (formulario de login en web o móvil). Ejemplo: usuario=`chofer1`, password=`secreto123`.

2. **Solicitud de Login:** La aplicación cliente envía una solicitud `POST /api/v1/auth/login` al servidor con las credenciales ingresadas (ver sección 8.3). Esta comunicación se realiza sobre HTTPS para seguridad.

3. **Verificación en Backend:** El controlador de autenticación (o filtro de Spring Security) recibe la solicitud. Utiliza el servicio de usuarios para cargar al usuario por nombre y compara la contraseña proporcionada con la almacenada (aplicando hash). Si la comparación falla (usuario no existe o contraseña incorrecta), devuelve error 401. Si es exitosa:

4. Genera un JWT usando JwtService. Se pasan los datos del usuario (id, rol, username)[34] y se firma con la clave secreta.
5. Opcionalmente registra la hora de login (se podría guardar en User último login, pero no implementado actualmente).

6. **Respuesta con Token:** El backend responde 200 OK con el token JWT en el cuerpo (y posiblemente info adicional como expiración y rol, como se mostró antes). El cliente recibe este JSON.

7. **Almacenamiento del Token (Cliente):** La aplicación extrae el token del JSON. En el caso de la app móvil, lo guarda en AsyncStorage para persistencia[36]. En la app web, se podría guardar en localStorage o en un cookie seguro (en esta versión probablemente localStorage).

8. **Establecer Sesión Aplicación:** Ahora la aplicación considera al usuario "logueado". En la web, redirige al dashboard principal. En la app móvil conductor, navega a la pantalla principal (p. ej., mapa de ruta). El token se coloca en un contexto de Auth (por ejemplo, AuthProvider en React context) junto con datos del usuario.

9. **Usar el Token en siguientes peticiones:** A partir de aquí, cada petición que el cliente haga al API incluirá el header `Authorization: Bearer <JWT>`. Esto es transparente para el usuario. Por ejemplo, si la app web carga la lista de vehículos, hace `GET /api/v1/vehicle` con el token en la cabecera.

10. **Validación del Token (cada request):** El filtro `JwtAuthenticationFilter` en el backend intercepta la petición entrante y detecta el header Authorization. Extrae el token[37], valida su firma y obtiene el username del claim[38]. Carga el UserDetails y lo coloca en el contexto de seguridad de Spring (SecurityContext) indicando que la petición está autenticada como ese usuario[39]. Luego la petición prosigue hasta el controlador específico.

11. **Acceso a Recursos Protegidos:** Dado que el token es válido, el usuario podrá acceder a los endpoints permitidos para su rol. Por ejemplo, un ADMIN podrá obtener la lista de usuarios, un DRIVER podría (en un futuro) acceder a un endpoint específico de conductor, etc. Si intentara acceder a algo no autorizado (no implementado granularmente ahora), se le negaría con 403.

12. **Expiración/Logout:** Si el usuario cierra sesión manualmente (clic en "logout"), la aplicación borra el token almacenado[40] y redirige a la pantalla de login. Si el token expira sin logout manual, el próximo request que haga resultará en 401 (el filtro detectará expiración). En ese caso, la app detectará el 401, limpiará cualquier rastro de sesión y pedirá login nuevamente.

Este flujo garantiza que las credenciales viajen solo una vez (en login) y luego el token JWT maneje la autenticación de forma segura y eficiente. Como no hay sesiones de servidor, el sistema es escalable (cualquier instancia de backend puede validar el token sin compartir estado).

10.2 Flujo de Rastreo en Tiempo Real

Este es el flujo crítico donde se actualiza y visualiza la posición de los buses en tiempo real para los usuarios. Involucra la app de conductor, el broker/servidor de mensajes y la app de pasajero o panel admin.

1. **Inicio del seguimiento (Conductor):** Cuando un conductor comienza su ruta del día, inicia la app móvil de conductor e inicia sesión (siguiendo el flujo 10.1). Después de login, la app de conductor comienza a publicar su ubicación:
2. La app solicita permisos de ubicación y activa los servicios de geolocalización (GPS) del dispositivo.

3. Se configura para obtener coordenadas cada intervalo de tiempo (ej. cada 5 segundos) usando `expo-location` u otra API.
4. Cada vez que obtiene una nueva coordenada (lat, lng), la app envía esa información al sistema. Aquí hay dos posibles subflujos: a. **Vía MQTT:** La app publica un mensaje en el tópico MQTT correspondiente, e.g. `urbantracker/vehicle/10/location`. El mensaje contiene el ID del vehículo y las coordenadas actuales, p.ej.: `{"vehicleId": 10, "lat": 4.6097100, "lng": -74.0817500, "time": "2025-10-01T14:05:30Z"}`. b. **Vía Socket.io/HTTP:** Alternativamente, si se hubiese implementado, la app podría invocar un endpoint REST del backend como `POST /api/v1/location` con el cuerpo de coordenadas, o emitir un evento WebSocket a un servidor Node/Socket.io. (En nuestra arquitectura, asumiremos MQTT para aprovechar la eficiencia).
5. La app de conductor también puede mostrar en su UI que el tracking está activo (p.ej., un indicador "Enviando ubicación..." con cada update). Si el GPS está apagado o hay error, muestra mensajes de troubleshooting (como guiar a activar GPS, ver TROUBLESHOOT doc).

6. **Recepción en backend/broker:** El mensaje de ubicación enviado entra al sistema:

7. Si es MQTT: El broker Mosquitto recibe el publish en `urbantracker/vehicle/10/location`. El backend UrbanTracker puede estar suscrito a los tópicos relevantes (p.ej. a `urbantracker/vehicle/+/location` wildcard) para procesar todas las ubicaciones entrantes. Alternativamente, los clientes suscriptores (ver paso 3) recibirán directamente del broker.
8. Si es HTTP: El backend recibe la petición, la valida (token del conductor), extrae la data y luego internamente la reenvía a los subscriptores (por ejemplo, podría usar una lista de WebSocket sessions de usuarios suscritos a ese vehículo/ruta y enviar a cada uno).
9. En cualquier caso, el sistema también podría almacenar la posición en la base de datos (por ejemplo, actualizar un campo "lastKnownLocation" del vehículo, o insertar en una tabla de histórico). Actualmente no se detalla en la implementación, pero es viable. Quizás actualice la entidad Vehicle en memoria o BD con las nuevas coords.

10. **Difusión a clientes (Pasajeros/Admin):** Los usuarios que necesitan ver la ubicación reciben la actualización:

11. **Panel Admin (Web):** La aplicación web administrativa probablemente tiene un dashboard con un mapa mostrando todos los vehículos o los de cierta ruta. Para recibir datos en tiempo real en la web, el navegador se conecta al servidor de mensajes. Caso Socket.io: la app web se conecta al endpoint de websocket del backend al iniciar. Caso MQTT: la app web podría usar un cliente MQTT via WebSocket para conectarse al broker Mosquitto (si este

permite conexiones WS) suscribiéndose a tópicos de interés. Un enfoque más sencillo es que el backend reciba MQTT y luego emita via Socket.io a la web. Suponiendo este último:

- El backend, al recibir la nueva posición del vehículo 10, emite un evento por Socket.io a todos los clientes suscritos a, digamos, la sala "vehicle-10" o "route-101" (según cómo se agrupe). El mensaje sería similar JSON con id vehículo y coords.
- El panel web, que estaba a la escucha de eventos, ejecuta la actualización: encuentra el marcador correspondiente a ese vehículo en el mapa y actualiza su posición en la interfaz. Esto ocurre casi instantáneamente (latencias ~ms). El administrador ve el icono del bus moverse en el mapa en vivo.

12. **Manejo de Desconexiones o Errores:** Si un conductor pierde conexión de red, el backend/broker dejará de recibir. Los clientes pueden manejar esto detectando *timeouts*: por ejemplo, si no se recibe nueva posición de un bus en X segundos, pueden marcarlo como desconectado. Cuando el conductor recupera conexión, la transmisión se reanuda. Si la app de conductor se cierra, el flujo se detiene (posiblemente el backend o admin marquen ese vehículo como fuera de línea tras un tiempo).

13. **Finalización de la ruta:** Cuando un bus termina su recorrido, el conductor puede detener el tracking (app de conductor podría tener un botón "Finalizar"). Al hacerlo, deja de enviar ubicaciones. El sistema podría opcionalmente registrar el fin de un Journey en la BD. Los clientes dejan de ver movimiento; podrían ver al bus estacionado en la última parada o removerlo del mapa.

Este flujo logra el **objetivo principal**: usuarios viendo buses en tiempo real. Los puntos críticos son la confiabilidad de la obtención de GPS (de ahí la sección de solución de problemas) y la infraestructura de mensajes. MQTT es eficiente con muchos clientes; Socket.io simplifica la comunicación con webs. UrbanTracker aprovecha ambos donde son más útiles.

Diagrama de Secuencia Simplificado: (Descripción textual)

```
ConductorApp --> Backend (via MQTT Broker): [Nueva posición lat,lng de
vehículo X]
Backend --> AdminWebApp: (Socket.io evento) [Vehículo X lat,lng]
Backend --> PasajeroApp: (Socket.io evento o MQTT) [Vehículo X lat,lng]
AdminWebApp --> Mapa: mover marcador Vehículo X a nueva lat,lng
PasajeroApp --> Mapa: mover bus de ruta a nueva lat,lng
```

(El diagrama asume backend relay; si PasajeroApp usa MQTT directo, el broker envía a PasajeroApp sin pasar por backend.)

10.3 Flujo de Gestión Administrativa (Alta de Vehículo)

Este flujo ejemplifica cómo un administrador utiliza el panel web para agregar un nuevo vehículo a la plataforma, asignarlo a una compañía y a un conductor.

1. **Acceso al módulo Vehículos:** El administrador navega en la web al apartado "Vehículos". La aplicación web realiza `GET /api/v1/vehicle` para mostrar la lista actual de vehículos. Esta petición incluye su token admin, el backend valida JWT[19] y responde con la lista de vehículos en JSON[61]. El navegador despliega una tabla con los vehículos existentes.

2. **Iniciar creación:** El admin hace clic en "Agregar Vehículo". Se muestra un formulario donde ingresa los datos: placa, marca, modelo, compañía (selecciona de un combo), estado (ej. activo), y conductor asignado (selecciona de lista de usuarios conductores disponibles).

3. **Enviar formulario:** Al confirmar, la web envía `POST /api/v1/vehicle` con un JSON en el cuerpo que contiene los campos rellenados (como descrito en 8.6). Incluye el token admin en cabecera. El backend recibe la solicitud en VehicleController (a implementar, supongamos que similar a Role/User creation):

4. Valida el token (autenticación ok).
5. Valida datos: que la placa no exista, que el usuario asignado exista y tenga rol DRIVER, etc. Esto se haría en VehicleService.
6. Crea la entidad Vehicle y la guarda via `IVehicle.save(...)`. Esto inserta en la BD el nuevo registro (gracias a JPA). Se propagan cascadas: quizás el assign de user/compañía ya existentes se manejan por sus IDs.
7. Devuelve una ResponseDTO de éxito[58] o los datos del vehículo creado (depende de implementación; podríamos suponer que devuelve el objeto creado).

8. **Actualizar UI:** La respuesta llega a la app web. En caso de éxito, se puede:

9. Mostrar un mensaje "Vehículo creado correctamente".
10. Actualizar la lista local de vehículos, añadiendo el nuevo sin recargar toda la página (o alternativamente volver a hacer GET vehicles para refrescar).
11. Cerrar el formulario modal.

12. **Asignación efectiva:** Ahora el nuevo vehículo está en el sistema. El conductor asignado fue enlazado: en la BD, el vehicle.user_id apunta al conductor dado. Si ese conductor tiene la app móvil, la próxima vez que inicie sesión podría ver que tiene ese vehículo asignado (aunque actualmente la app conductor no tiene un listado, podría en un futuro).

13. Si el vehículo está activo, también podría empezar a aparecer en los mapas. Por ejemplo, en la sección de admin, si filtra por la compañía del vehículo, ahora lo verá (aunque sin posición hasta que envíe datos).
14. En la base de datos, el registro en `vehicle` incluye su status y demás.

15. **Casos alternos:** Si la placa ingresada ya existía, el backend habría devuelto 409 Conflict. La web debe manejarlo mostrando un error "Ya existe un vehículo con esa matrícula". El admin corregiría y reintentaría. Si se ingresa un user id que no es rol DRIVER, probablemente el backend igual asigna (no tiene validación actual para eso salvo que podría hacer check manual). Debería idealmente rechazar con 400 "El usuario seleccionado no es un conductor válido". Esto se puede implementar en VehicleService.

Un flujo similar se sigue para otras entidades: - **Registrar nuevo Conductor:** El admin iría a módulo Usuarios, crearía un user con rol DRIVER, posiblemente asociándolo a una compañía. Luego podrá asignarlo a un vehículo. - **Crear Ruta:** En módulo Rutas, ingresa nombre y compañía, luego añade puntos (podría ser sub-flujo de cargar puntos en un mapa interactivo, o ingresar coordenadas manual). - **Asignar Vehículo a Ruta:** Esto no está muy explícito; podría implicar crear un Journey programado. Pero conceptualmente, un admin podría "despachar" un vehículo a una ruta, lo cual en sistema se reflejaría como crear un Journey con startTime futuro, etc.

10.4 (Opcional) Flujo de Consulta para Pasajero:
Si un pasajero utiliza la app sin registrarse, el flujo sería: 1. Abre la app, selecciona la ruta que le interesa ver. 2. La app hace `GET /api/v1/route` para listar rutas disponibles (pudiendo filtrar compañías si hay varias). 3. Usuario elige ruta 101. La app hace `GET /api/v1/route/101` para obtener detalles (incluye paradas, etc.) y `GET /api/v1/vehicle?route=101` (si existiera un filtro así) o recibe por realtime qué vehículos están en esa ruta. 4. Muestra el mapa con la ruta trazada (usando los RoutePoints) y los buses actuales (posiciones actualizadas en tiempo real con flujo 10.2). 5. El usuario observa en tiempo real hasta que cierra la app.

Este flujo muestra cómo varias llamadas API (rutas, puntos, etc.) se combinan con la capa de tiempo real para la experiencia de seguimiento.

Cada uno de estos flujos de negocio fue considerado en la arquitectura y diseño de UrbanTracker, asegurando que el sistema soporte los casos de uso claves de forma eficiente y segura.

11. Pruebaз y Validación

Para garantizar la calidad y correcto funcionamiento de UrbanTracker, se han contemplado diversas estrategias de pruebas, abarcando desde pruebas unitarias de

componentes individuales hasta pruebas integrales del sistema en condiciones reales.

11.1 Estrategias de Pruebas Unitarias (Backend)

El proyecto backend incluye un módulo de pruebas automatizadas usando JUnit 5 y Spring Boot Test. Un ejemplo mínimo es la prueba de carga de contexto (`UrbanTrackerApplicationTests`) que verifica que la aplicación arranca correctamente[93]. Sin embargo, es esencial expandir la suite de pruebas unitarias para cubrir la lógica de negocio en los servicios: - *Pruebas de Servicios:* Cada método crítico de los servicios (UserService, CompanyService, etc.) debería probarse con distintos escenarios. Por ejemplo, probar que `UserService.save` retorna error apropiado si el username ya existe, o que `CompanyService.save` valida correctamente el formato de NIT[94][50]. Para esto, se pueden utilizar mocks de los repos (usando Mockito) para simular distintas condiciones (e.g., `iUser.existsByUserName` devolviendo true/false). - *Pruebas de Repositorios:* Como se usan consultas custom (por ej., `IUser.getAll()` con JPQL[10]), conviene probar que dichas consultas funcionen y mapeen correctamente a DTOs. Estas pruebas se pueden hacer con SpringBootTest cargando una BD en memoria (H2) para validar la integración JPA. - *Pruebas de Controladores:* Se pueden usar MockMvc de Spring para simular peticiones HTTP a los endpoints y verificar las respuestas. Por ejemplo, hacer un POST a `/api/v1/public/user` con JSON válido y comprobar que retorna 200 y el mensaje "usuario guardado correctamente". También probar llamadas no autorizadas (sin token a endpoints privados) para asegurar que devuelven 401.

Actualmente, la cobertura de pruebas unitarias en dev es baja. Aumentarla es prioritario para evitar regresiones en lógica según evolucione el proyecto.

11.2 Pruebas de Integración y Sistema

Además de pruebas unitarias, se realizan: - *Pruebas manuales con Postman/ThunderClient:* El equipo utiliza herramientas como Postman para probar los endpoints REST manualmente. Se configuran colecciones de API (login, CRUD de entidades) e incluye los tokens JWT en las cabeceras para probar flujos completos. Por ejemplo, tras loguearse con un admin, probar crear un nuevo vehículo y luego listarlo. Estas pruebas aseguran que los distintos módulos funcionan en conjunto (BD + servicio + controlador + seguridad). - *Swagger UI:* La integración de springdoc OpenAPI permite probar los endpoints desde el navegador con la interfaz de Swagger. Se puede ingresar un token en la función "Authorize" y luego ejecutar operaciones para verificar resultados rápidamente. - *Pruebas de UI Web:* Dado que es una aplicación React, se realizan pruebas manuales de la interfaz en distintos navegadores (Chrome, Firefox). Se verifica que la navegación funcione, que las tablas carguen datos reales (indicando que las llamadas API están bien integradas), y que formularios de creación/edición validen entradas. Si algún comportamiento es crítico (ej: lógica compleja en un hook), se pueden escribir pruebas con Jest/React Testing Library, aunque por ahora no se han implementado en dev. - *Pruebas en Dispositivos Móviles:* Se hacen pruebas tanto en emulador Android como en dispositivos físicos: -

Emulador Android: Útil para pruebas rápidas de la app de conductor, simulando movimiento de GPS. Por ejemplo, usando los controles de ubicación del emulador para enviar coordenadas de prueba y verificar que la app las captura (como sugiere la guía de solución de problemas)[95]. También se monitorean logs con `npx react-native log-android` para ver que la app no arroje errores[96]. - **Dispositivo Físico:** Fundamental para pruebas de GPS real. Se instala la app Expo en un teléfono Android, se ejecuta la app de conductor en modo desarrollo y se mueve físicamente o se simula movimiento para ver si las coordenadas se envían y aparecen en el sistema. Según la guía, es importante probar en exteriores para buena señal GPS[97]. - **Escenarios de Permisos:** Se prueba qué ocurre si el usuario deniega permisos de ubicación en la app de conductor: la app debe manejarlo (mostrar error "Permisos denegados", como en los estados definidos)[98]. También, que pasa si el GPS del dispositivo está apagado: la app muestra el mensaje "No valid location provider" y guía a encenderlo[22][51]. - *Pruebas de Flujo Completo:* Se ensayan casos de uso de principio a fin. Ejemplo: Crear un nuevo conductor -> usar sus credenciales en la app móvil -> ver cómo aparece en admin. Otro: Iniciar app de conductor -> moverlo -> observar en app de pasajero el movimiento en tiempo real. Estos ensayos garantizan que la integración entre componentes (apps, backend, broker) es correcta.

11.3 Cobertura y Herramientas

- La cobertura de código (test coverage) idealmente debería medirse con herramientas como Jacoco para backend. Actualmente, con pocas pruebas, la cobertura es baja; a mediano plazo, se buscará cubrir al menos 70% del código de servicios y utilidades. - Para la app web, se pueden integrar pruebas E2E con herramientas como Cypress para simular un usuario real: haciendo login en la UI, navegando, creando entidades y verificando que aparecen los cambios. Esto no se ha implementado aún, pero es una recomendación futura. - En la apps móvil, Expo no tiene un framework de pruebas UI integrado, pero se puede usar Appium o Detox para tests end-to-end en React Native. Dado el foco en tiempo real, muchas pruebas móviles consistirán en manual + monitoring.

11.4 Pruebas de Rendimiento (Consideraciones)

Dado que se manejan componentes en tiempo real, se planea realizar pruebas de carga: - Simular múltiples dispositivos conductor enviando ubicaciones simultáneamente (por ej., 50 conductores enviando cada 5s) y medir cómo soporta el broker/backend. Herramientas: scripts con paho-mqtt o k6 (send HTTP if using HTTP approach). - Medir latencia desde envío de posición hasta recepción en cliente (posiblemente log timestamps in message and compare). - Pruebas de volumen en BD: insertar gran cantidad de registros (usuarios, viajes) para ver que las consultas siguen respondiendo rápido (quizá con EXPLAIN ANALYZE en PostgreSQL para rutas intensivas).

11.5 Validación de Seguridad

Se realizan pruebas específicas para asegurar la robustez de la seguridad: - Intentar acceder a recursos sin token o con token inválido (esperar 401). - Intentar roles

incorrectos (si implementado en el futuro). - Probar vectores XSS/SQLi: enviar payloads maliciosos en campos (script tags en nombres, etc.) y verificar que o bien son filtrados por validación o no causan comportamientos indebidos. La validación regex en servicio (ej: solo letras y espacios para roles[45], etc.) mitiga esto.

11.6 Uso de Entorno de Pruebas

Para no afectar datos reales, se suele probar en entornos separados: - Base de datos H2 en pruebas unitarias/integración. - Un ambiente de *staging* con configuración equivalente a producción para pruebas finales antes de desplegar (en caso de un despliegue real). - En Expo, usar canales release candidate para probar builds en testers antes de publicar a stores.

En conclusión, aunque la rama dev muestra solo pruebas básicas, el plan de pruebas abarca niveles múltiples. Conforme el proyecto madura, se irá incrementando la automatización de pruebas. Mientras tanto, las pruebas manuales guiadas por casos de uso y la atenta verificación de logs (tanto del servidor como de las apps) han sido fundamentales para depurar y validar funcionalidades durante el desarrollo.

12. Compilación y De3pliegue

En esta sección se detallan los pasos y consideraciones para compilar los distintos componentes de UrbanTracker y desplegarlos en un entorno de producción. También se discuten aspectos de integración continua (CI/CD) y versionamiento del software.

12.1 Modo Desarrollo

Durante el desarrollo, se utilizan las herramientas y comandos descritos en la sección 4 para ejecución en caliente. Algunos recordatorios clave: - El backend se ejecuta con `mvn spring-boot:run`, lo cual recarga la aplicación en caso de cambios si se activa DevTools (no mencionado, pero se puede agregar Spring Boot DevTools para auto-restart). En desarrollo, suele correr en puerto 8080. - La app web con `npm run dev` en Next.js recarga los cambios de React en tiempo real. Está en puerto 3000 por defecto. - Las apps Expo corren con `expo start`, donde el modo *development* habilita recarga rápida (Fast Refresh) en el dispositivo/emulador al guardar cambios de JavaScript. Esto acelera mucho la iteración en UI móvil. - El broker MQTT (si usado local) se lanza con `mosquitto` (si está instalado como servicio, puede estar ya corriendo en 1883). - Base de datos: en desarrollo se suele apuntar a una instancia local de PostgreSQL. Se puede usar Docker para levantar rápidamente una BD:

```
docker run --name urbantracker-db -e POSTGRES_PASSWORD=postgres -e
POSTGRES_DB=urbantracker_db -p 5432:5432 -d postgres
```

Luego conectar con user postgres. Para entorno dev, esto simplifica no interferir con otras BDs.

12.2 Generación de Builds (Producción)

Una vez que se desea desplegar UrbanTracker en producción, se deben generar *builds* optimizadas de cada componente:

- **Backend (Spring Boot Jar):** Usando Maven, ejecutar `mvn clean package`. Esto producirá un archivo JAR ejecutable (por ejemplo `UrbanTracker-0.0.1-SNAPSHOT.jar` en `Backend/target/`). Ese JAR contiene el servidor embebido y todo lo necesario. Se puede ejecutar con `java -jar UrbanTracker-...jar`. Para producción, conviene configurar variables de entorno/propiedades externas para DB, puerto, etc., en lugar de usar las de desarrollo.
- *Configuración Prod:* por ejemplo, usar `SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME`, etc., o tener un `application-prod.properties`.
- Se puede especificar un puerto diferente con `server.port=80` si se quiere que corra en 80/443 detrás de un proxy, etc.
- Considerar habilitar logs más detallados al inicio, pero menos verbosos luego (prod usually info level).
- ***Dockerización:*** Una buena práctica es crear un Dockerfile para el backend. No provisto en dev, pero se puede basar en openjdk:17-alpine, copiar el jar y set entrypoint.
- **Aplicación Web (Next.js static build):** Para desplegar la app web de administración, se debe crear la versión de producción:
- Ejecutar `npm run build` en `Web-Admin/`. Next.js generará la aplicación optimizada. Dado que se usa el App Router en Next13, la build genera una aplicación Node (no puramente estática). Normalmente, tras build, se usa `npm run start` para lanzar la web en modo producción (sirviendo las páginas pre-renderizadas).
- Alternativamente, se puede desplegar en Vercel u otra plataforma que acepte Next directamente. Pero supongamos servidor propio: se incluiría Node.js en el servidor, instalar dependencias (o copy node_modules as part of build process), luego run the start script.
- Configurar variables como la URL base de la API (en producción, la app web debe saber dónde está el backend). Esto se puede hacer via variables de entorno públicas de Next (NEXT_PUBLIC_API_URL). En dev podía ser localhost:8080, en prod quizás un dominio o microservicio diferente.
- ***Dockerización:*** Podría dockerizar la web admin también, usando multi-stage: stage1 compile (node:18-alpine) then stage2 run (perhaps using a Node runtime or exporting to static if possible).
- **Aplicaciones Móviles (APK/IPA):** Para distribuir la app móvil fuera del entorno dev (Expo Go), se necesita generar paquetes nativos:
- Utilizando **Expo EAS (Expo Application Services)**, se puede crear builds automáticos. Configurar eas.json con perfiles (development, production).

Después: `eas build -p android --profile production` generaría un APK/AAB de Android listo para Play Store.

- Alternativamente, se puede hacer un "expo prebuild" y luego abrir en Android Studio, pero lo recomendado es EAS.
- Antes de build, actualizar `app.json` para producción: poner el `package` Android (ej. com.miempresa.urbantrackerconductor), el nombre de la app, iconos finales, etc. Este ya está configurado en dev con placeholders (`com.anonymous.UrbanTracker`)[99][100], se debe reemplazar con identificadores reales.
- Pruebas finales: instalar el APK en dispositivos reales y verificar conectividad con la API en la URL de producción (posiblemente cambiar la base URL en el código o usar un .env prod).
- **Broker MQTT:** En producción, se puede usar una instancia de Mosquitto o algún servicio en la nube. Configurar autenticación en el broker (usuarios/contraseñas o certificados) si es accesible públicamente, para que solo componentes de UrbanTracker lo usen. Ubicarlo en un servidor con puerto 1883 abierto para apps (o 8883 TLS). Si se usan websockets MQTT, habilitar en config de Mosquitto el listener de 9001 (por ejemplo).
- **Servidor y Balanceo:** Decidir despliegue final: se puede desplegar todo en un solo servidor (monolítico): Backend Java + Mosquitto + Node (Next) + serve static, etc. O separarlos (backend en un servicio, web static hosteada en S3/CloudFront, MQTT en otro). Dependiendo de escalabilidad, modular.
- **Dominio y SSL:** Registrar un dominio para el servidor. Configurar un proxy Nginx/Apache para manejar SSL (443) y direccionar las peticiones:
- `https://urbantracker.com/api` -> backend (internamente :8080),
- `https://urbantracker.com/` -> app web (Next start at 3000 or serve static built version),
- `wss://urbantracker.com/socket.io` -> route to backend (if using socket.io on same port or separate Node service).
- Alternatively, host the Next app on same Express as API (but en este caso son separados tech, así que mejor proxy).
- **Environment Variables en Prod:** Asegurarse de establecer:
- `SPRING_PROFILES_ACTIVE=prod` (si se tiene profile prod),
- Variables de DB (host, user, pass),
- JWT secret (no usar el de dev en prod, generar uno nuevo y pasarlo por env),
- API base URL in Next (maybe baked at build or set via runtime config if SSR),
- Any API keys (Google Maps API key if needed in front-end maps).
- **Migraciones de Base de Datos:** Actualmente se usa `ddl-auto=update` para auto-crear tablas. En producción, es mejor usar migraciones controladas (Flyway/Liquibase) para versionar el esquema. Por ahora se confiaría en

update para primer despliegue, pero en evoluciones futuras conviene integrar Flyway and include SQL migration scripts.

12.4 Integración Continua (CI/CD) y Control de Versiones

Para agilizar el ciclo de desarrollo y despliegue, se pueden implementar prácticas de CI/CD: - *Repositorio Git:* El proyecto usa GitHub para control de versiones (repo AFSB114/UrbanTracker). Se siguen ramas **dev** (desarrollo) y **main** (producción). Los cambios se integran en dev, se prueban, y periódicamente se fusionan a main una vez estables[101]. Este flujo se observa en commits de merge dev->main y viceversa[102]. - *GitHub Actions:* Se puede configurar un flujo de CI en GitHub Actions para: - Compilar el backend y ejecutar pruebas en cada push (asegurando que no se rompa nada). - Ejecutar `npm run build` para la app web y (futuro) correr pruebas de front. - Posiblemente, construir imágenes Docker y empujarlas a un registry. - Incluso, con Action, hacer trigger de EAS builds para mobile (Expo tiene actions or use eas-cli). - *Despliegue Continuo:* Según la infraestructura, se puede automatizar despliegues: - Por ejemplo, si usando Kubernetes, tras push a main un pipeline podría actualizar los deployments (usando kubectl or helm). - O en un servidor sencillo, usar GitHub Actions SCP/SSH para copiar artefactos y reiniciar servicios. - También se puede integrar con services como Heroku, Vercel (para la web, ya que Next podría hostearse en Vercel hooking main branch). - *Versionamiento Semántico:* UrbanTracker podría adoptar versionado semántico: v1.0.0, v1.1.0, etc. De momento está en "0.0.1-SNAPSHOT"[103]. Cuando se haga un release formal, se actualiza en pom.xml la versión, y se taguea en Git una release. Las apps móviles versiones ya se indicaron en app.json como "1.0.0" para la primera release[104], y deberán incrementarse con cada publicación (Android requiere versionCode incremental). - *Manejo de Configuraciones:* Para CI/CD es útil no hardcodear variables sensibles. Utilizar secretos de GitHub Actions para credenciales DB, JWT secret, etc. Y templating o env injection en los archivos de config de producción. - *Backups:* Planificar backups de la base de datos en producción (no es CI/CD pero es parte de mantenimiento). Podría usarse pg_dump programado.

Con CI/CD implementado, el ciclo sería: un desarrollador mergea a main -> pipeline construye C prueba -> si todo OK, despliega a staging/prod. Esto reduce errores de despliegue manual.

Consideraciones de Versiones de Dependencias en Prod: - Asegurar usar la misma versión de Node/Java en producción que en dev (evitar "funciona en dev, rompe en prod" por versiones). - Monitorizar los logs tras despliegue para cualquier excepción no vista en dev (por carga, datos reales, etc.).

En resumen, la compilación y despliegue de UrbanTracker implican construir componentes separados y coordinarlos en un entorno unificado. Con prácticas de CI/CD se busca minimizar intervenciones manuales y asegurar que cada actualización sea confiable. Siguiendo estos lineamientos, el equipo puede iterar

rápidamente y llevar nuevas funcionalidades a producción de forma segura, manteniendo la estabilidad del servicio para los usuarios finales.

13. Reзolución de Problemaз

A continuación, se enumeran algunos problemas comunes que pueden surgir al ejecutar o usar UrbanTracker, junto con sus posibles causas y soluciones recomendadas. Esta guía de troubleshooting ayuda a diagnosticar rápidamente incidencias en los distintos componentes (móvil, web, backend, infraestructura).

- **La aplicación móvil de conductor muestra error `"There is no valid location provider available."`:** Este mensaje indica que no hay proveedores de ubicación habilitados en el dispositivo[22]. En dispositivos Android, significa que el GPS está desactivado o los permisos no fueron otorgados correctamente. **Solución:**
- Verificar que el dispositivo tenga activada la ubicación (GPS). En Android, ir a *Ajustes* > *Ubicación* y asegurarse de que esté en "Alta precisión"[105]. En un emulador, habilitar la casilla "Use location".
- Asegurarse de haber concedido los permisos de ubicación a la app. En Android 10+, verificar en *Ajustes* > *Apps* > *UrbanTracker Driver* > *Permisos* que *Ubicación* esté en "Permitido".
- Si el problema persiste en emulador, se pueden enviar coordenadas manualmente: en Android Studio emulator, usar la pestaña *Location* para simular un GPS fix[106]. También se pueden usar comandos ADB para habilitar providers[107].

- **La app móvil (conductor) no conecta con el servidor (no carga datos / no envía ubicación):** Posibles causas: la URL/puerto del backend está mal configurada en la app, o problemas de red (firewall, conexiones locales). **Solución:**

- Confirmar la configuración de la URL base de la API en la app. En modo desarrollo, si se usa Expo LAN, la URL podría ser algo como `http://192.168.x.y:8080`. Asegurarse de que la IP corresponda al PC donde corre el backend. *Recuerda:* `"localhost"` desde el móvil no funciona, debe ser la IP del PC en la misma red.
- Probar el endpoint manualmente desde el dispositivo: por ejemplo, instalar una app tipo Postman en el teléfono e intentar hacer un GET a `http://<IP-PC>:8080/api/v1/public/user`. Si falla, es de red.
- Verificar firewall: en Windows, permitir java.exe en el puerto 8080; en Linux, iptables/ufw permitir 8080, etc.

- Si el dispositivo está en red celular y el backend en tu PC no es accesible públicamente, la app no podrá llegar. Usa el modo "Tunnel" de Expo (`expo start --tunnel`) o conecta el dispositivo a la misma WiFi.
- En producción, verificar que el dominio o IP público esté bien y que el certificado SSL esté configurado (si la app intenta https). Un truco: habilitar logs en la app para imprimir la URL de fetch que intenta, y cualquier error de fetch (código de error de red).

- **Los mapas no se visualizan (mapa en blanco o errores de API keys):** UrbanTracker usa Google Maps y Mapbox. Si no aparecen los mapas:

- Asegurarse de tener las API keys correspondientes. Por ejemplo, para Google Maps en Expo se necesita proveer la clave de Maps SDK para Android e iOS en `app.json` (campo `expo.android.config.googleMaps.apiKey`). Si se ve un error de "Google Maps API key is required", obtener una desde Google Cloud y agregarla.
- En caso de Mapbox, si se utiliza MapBoxGL, se requiere un token de acceso Mapbox. Este token normalmente se pasa en la inicialización del mapa. Verificar que se haya configurado correctamente en `MapBox.tsx` o en variables de entorno.
- Revisar la consola/log: en web, puede salir un mensaje de Maps API error (p.ej. "Invalid API key" o "Referer not allowed"). En mobile, revisar logs expo para errores de mapa.
- Si la clave está, asegurar que la plataforma está habilitada en el dashboard de Google (p.ej. Android SDK enabled).
- Mapbox: Generar un token en account.mapbox.com y ponerlo. Si MapBoxMQTT.tsx lo usa, revisar que no esté vacío.

- **Latencia de red:** Si se nota retardo, puede ser latencia celular. En ese caso, poco que hacer más que asegurar buena cobertura. MQTT por 3G/4G suele ser rápido, pero si la red está saturada, habrá lag.
- **Frecuencia de envío:** Confirmar que la app conductor esté enviando con la frecuencia esperada. Podría estar configurada cada 5s; si se quisiera más rápido, se puede bajar intervalo, pero a costo de batería.

- **Error 500 en alguna operación del API (ej., al crear entidad):** Ver los logs del backend (stack trace). Comúnmente podría ser error de integridad (p. ej. violación de UNIQUE que no se manejó con catch). Mejorar el servicio para que detecte y devuelva un ResponseDTO adecuado. Mientras tanto, la solución es corregir la causa raíz (datos duplicados, null inesperado). En desarrollo, agregar bloques try-catch en los servicios si no están, para retornar ResponseDTO con error en vez de lanzar excepción no controlada[53].

- **La aplicación web no carga (pantalla en blanco o error de build):** Si tras desplegar la web admin solo se ve página blanca:

- Abrir la consola del navegador y buscar errores de JavaScript. Un error común es no poder conectar con la API (si la web hace fetch a `/api/v1/...` relativo y el dominio no coincide o CORS no configurado). Configurar correctamente la URL base o CORS.
- Si es un error de React (stacktrace in console), puede ser por una variable de entorno faltante en build. Revisar que en producción se esté pasando NEXT_PUBLIC_API_URL y cualquier otra necesaria.
- Verificar que se subieron todos los archivos estáticos (si se usa `next export`, checar que el output `out/` tenga pages, etc.).
- Si se sirvió la app en path distinto (ej /admin/), Next config basePath debe reflejarlo. Si no, rutas romperán.

- **Problemas de permisos en producción (e.g., 403 en endpoints esperados):** Si más adelante se añaden restricciones de roles en endpoints y un usuario autorizado recibe 403:

- Verificar que el token JWT realmente contiene el rol correcto y que el mecanismo de autorización está configurado bien (por ejemplo, usar `hasAuthority('ADMIN')` en Spring Security coincide con cómo se cargan roles – en este proyecto se cargan roles como nombre simple via GrantedAuthority[11], así que debería funcionar).
- También comprobar reloj del sistema: tokens JWT tienen expiración basada en tiempo; si el server y client difieren mucho en hora, se podrían rechazar prematuramente. Mantener sincronizado (NTP).

- **Uso elevado de CPU/RAM en el servidor:** Si se observa que el backend consume demasiados recursos:

- Revisar que no haya bucles intensivos en la lógica (no parece haber).
- Podría ser por muchas conexiones abiertas (websockets, etc.). Ajustar parámetros de Socket.io if needed (ej: ping-pong intervals).
- Posible memory leak: el uso de cascade ALL en relaciones OneToMany sin cuidado podría cargar muchos objetos en memoria. Observar si las consultas devuelven demasiados datos (ej: UserService.getAllUsers usa una JPQL que devuelve List<Object[]> para mappear a DTO; eso está bien).
- De ser necesario, asignar más memoria (flag -Xmx en JVM).
- Monitorizar con JProfiler o similar si se sospecha de fugas.

- **Errores en envío/recepción MQTT (certificados):** Si se implementa TLS en MQTT, la app podría requerir configuración adicional para confiar en cert self-signed. Mejor usar Let's Encrypt válido. En caso de error "SSL handshake

failed", verificar certificados en el dispositivo (posiblemente se necesite incluir CA).

En general, la resolución de problemas se apoya en: - **Logs detallados:** Mantener logging adecuado. Por ejemplo, en dev se pueden tener logs DEBUG para SQL de Hibernate, etc. En producción, logs informativos con suficiente contexto. - **Herramientas de monitoreo:** Emplear herramientas como logcat (Android), DevTools (web), Wireshark/tcpdump (red) para diagnosticar problemas de comunicación. - **Documentación de terceros:** Consultar documentación de Expo/React Native ante errores nativos, documentación de Spring si hay errores de configuración, etc. Por ejemplo, el error de localización fue abordado mejorando la configuración de la librería react-native-location según su README[108].

La tabla a continuación resume algunos problemas y soluciones rápidas:

Problema	Causa posible	Solución
"No valid location provider" en app driver	GPS desactivado, sin permisos	Activar GPS; otorgar permisos; ver [89].
App móvil no conecta a API	URL backend incorrecta; red	Usar IP en la URL; misma red o túnel; abrir puertos.
Mapas no cargan	Falta API Key; token inválido	Configurar clave Google Maps en app; token Mapbox.
Sin actualizaciones en tiempo real	Conductor offline; fallo broker	Verificar conexión conductor; probar broker MQTT; revisar config Socket.
Error 401 en endpoints privados	Token ausente/expirado	Login nuevamente; revisar expiración y hora sistema.
Error 409 al crear recurso	Datos duplicados (unique)	Cambiar valor (ej. otro username/placa); o eliminar existente conflictivo.
Web admin en blanco tras deploy	Falla en build o config	Revisar consola navegador; corregir URL API; build Next adecuadamente.
Crash de app móvil al iniciar	Config incorrecta; bug código	Ejecutar con `expo start --debug`, leer stack trace; arreglar origen (posible variable indefinida, etc.).
Backend Memory Leak (hipotético)	Manejo ineficiente de entidades	Usar paginación en listas grandes; evitar cascade excesivo; probar con VisualVM.
Conductor no puede hacer logout (token persiste)	Bug en app (no borra AsyncStorage)	Confirmar AuthService.logout es llamado; verificar logs de logout; posible llamada incorrecta.

Problema	Causa posible	Solución
Token JWT no válido inmediatamente tras crearlo	Dif. de tiempo server/cliente	Sincronizar relojes; verificar timezone.

Siguiendo estas guías, la mayoría de las incidencias conocidas pueden resolverse o incluso prevenirse configurando correctamente los entornos. Ante problemas nuevos, se recomienda aislar el componente (¿es backend, red o cliente?), reproducir el escenario en un entorno de prueba, y consultar logs junto a la documentación pertinente.

14. Mantenimiento y Monitoreo

El mantenimiento de UrbanTracker abarca tanto la salud operativa del sistema en producción como las actualizaciones evolutivas del software. A continuación, se listan prácticas y herramientas para mantener el sistema estable y monitorizado:

Mantenimiento Preventivo:
- **Actualización de Dependencias:** Periódicamente revisar y actualizar las librerías usadas. Mantener Spring Boot en su última versión de parche (para seguridad), actualizar Expo SDK cuando sea estable, etc. Antes de actualizar en producción, probar en desarrollo que no haya breaking changes. - **Limpieza de Datos:** Con el tiempo, la base de datos podría crecer (especialmente si se almacenan históricos de posiciones o journeys). Definir una política de retención: por ejemplo, mantener sólo 6 meses de registros de viaje detallados, o archivar datos antiguos en otra tabla/backup. Implementar procedimientos (scripts SQL o endpoints admin) para eliminar o archivar datos antiguos. - **Backups Regulares:** Configurar backups automáticos de la base de datos PostgreSQL (diarios o semanales según intensidad de cambios). Probar la restauración de backups en un entorno de staging para asegurarse de su integridad. También exportar configuración de Mosquitto si se usa (aunque su config es estática, los datos en broker son volátiles). - **Documentación y Conocimiento:** Mantener este manual técnico actualizado a medida que el sistema evolucione. Si se agregan nuevos módulos (ej. notificaciones), documentarlos. Además, tener un README operativo para los comandos comunes (arranque, despliegue). - **Mantenimiento de Certificados:** Si se usan certificados SSL (para el dominio web o MQTT TLS), llevar control de su expiración y renovarlos (Let's Encrypt cada 90 días, se puede automatizar).

Monitorización en Producción:
Implementar herramientas para vigilar el rendimiento y disponibilidad del sistema: - **Logs Centralizados:** Configurar el backend para rotar logs (por ejemplo, diario) y almacenarlos. Usar una solución como ELK (Elasticsearch + Logstash + Kibana) o un servicio cloud para recolectar logs de la app web, backend y quizá logs relevantes de

Mosquitto. Esto permite búsquedas cuando hay incidentes. - **Monitoreo de Servidor:** Usar herramientas como Grafana+Prometheus, New Relic, Datadog, etc., para monitorear métricas: uso de CPU, memoria y disco del servidor; métricas de la JVM (garbage collection, threads), conexiones activas; métricas de PostgreSQL (número de conexiones, slow queries). Spring Boot puede integrar Actuator endpoints para exponer algunas métricas fácilmente. - **Monitoreo de Aplicación:** - Para la parte móvil, se puede integrar un servicio de crash reporting (por ejemplo, Sentry, Firebase Crashlytics) dentro de las apps. De esta manera, si ocurre un crash en un dispositivo de usuario, se reporta automáticamente con el stack trace, ayudando a detectar y arreglar bugs que en testing no aparecieron. - Para la web, igualmente Sentry para capturar excepciones de JavaScript runtime en clientes. - **Uptime Monitoring:** Configurar un ping regular al endpoint de salud del backend (por ejemplo `/actuator/health` si se habilita, o `/api/v1/public/user` como dummy) desde un servicio externo (Pingdom, UptimeRobot). También monitorizar la URL MQTT si posible (difícil a menos que se tenga un cliente sintético). - **Alertas:** Establecer alertas que notifiquen al equipo si: el servidor se cae, uso de CPU > X% por Y tiempo, memoria casi llena, no se han recibido mensajes de ubicación en X horas (lo cual podría indicar problema en trackers), etc. Las alertas pueden ser vía email, Slack, SMS según las herramientas usadas. - **Manejo de Errores en Producción:** Determinar pautas para responder a incidentes: - Si el backend lanza excepciones no controladas repetidamente (ver en logs), plan de hotfix rápido. - Si el broker MQTT se cierra (poco probable), tener un script de reinicio supervisado (usando systemd service con Restart=always o docker auto-restart). - En caso de sobrecarga (ej. demasiados usuarios real-time), planear escalar horizontalmente: se podría tener múltiples instancias de backend y usar un Redis o similar para coordinar websockets (Socket.io require sticky sessions or pub-sub backend). Para MQTT, un broker cluster si hiciera falta. - **Seguridad Activa:** Además de monitorear performance, monitorizar seguridad: - Revisar logs de acceso en busca de patrones sospechosos (intentos de login repetidos -> podría implementar rate limiting en endpoints de auth). - Mantener actualizado el sistema operativo del servidor y los paquetes para cerrar vulnerabilidades conocidas. - Realizar periódicamente scans de vulnerabilidades (usando OWASP ZAP, etc., contra la API y la web) y resolver hallazgos.

Proceso de Deploy Seguro:
Para minimizar downtime en actualizaciones: - Considerar implementar despliegue azul/verde o canary: tener una segunda instancia donde lanzar la nueva versión, probarla, luego rotar tráfico. Con Kubernetes/containers esto es más fácil. En un solo servidor se puede detener la app por un momento, pero para high availability se piensa en instancias duplicadas. - Hacer despliegues en horarios de menor uso (ej. madrugada) si hay que reiniciar servicios. - Notificar a usuarios (sobre todo admins) de mantenimientos programados.

Soporte y Soporte Post-Deploy:
- Establecer un canal de soporte (como se indica en Contacto) para que usuarios reporten problemas. Documentar las incidencias comunes (FAQ) les ayudará a los

admins locales a resolver cosas simples (ej. "No veo mi bus" -> posiblemente conductor no conectado). - Mantener un registro de cambios (changelog) para que el equipo sepa qué versiones introducen qué mejoras o cambios de comportamiento (útil al debuggear: "ah, en v1.2 cambiamos cómo se envía la localización").

Optimización Continua:
- A partir de la monitorización, detectar cuellos de botella: ej, si la consulta de lista de usuarios es lenta con muchos registros, quizá paginarla. - Mejorar la experiencia de tracking: si se detecta que mandar cada 5s es mucha carga, pero 10s es suficiente, se podría ajustar en config sin redeploy (incluso hacer la frecuencia configurable centralmente a futuro). - Actualizar las apps móviles con feedback de usuarios (p. ej., agregar notificación de "Bus cerca" - seria feature futura, pero implicaría push notifications).

En síntesis, el mantenimiento exitoso de UrbanTracker requiere una combinación de **prevención**, **vigilancia activa** y **capacidad de reacción**. Con las herramientas y prácticas mencionadas, el equipo técnico puede asegurar que la plataforma se mantenga confiable y eficiente en el tiempo, brindando un servicio de calidad tanto a los administradores como a los usuarios finales que dependen de la información en tiempo real.

15. Referencia3 y Contacto

15.1 Documentación Técnica de Referencia
- **Java s Spring Boot:**
- Spring Boot Reference: spring.io/docs/spring-boot
- Spring Security C JWT Guide: spring.io/guides/topicals/spring-security-architecture (buscar JWT)
- JPA/Hibernate Documentation: docs.jboss.org/hibernate
- **Base de Datos PostgreSQL:**
- PostgreSQL Official Docs: postgresql.org/docs
- JDBC Postgres Driver: github.com/pgjdbc (release notes, issues)
- **React s Next.js (Web Admin):**
- React Documentation: reactjs.org/docs
- Next.js Documentation: nextjs.org/docs
- Tailwind CSS Docs: tailwindcss.com/docs
- shadcn/UI (Radix components) if used: ui.shadcn.com
- **React Native s Expo (Apps Móviles):**
- React Native Docs: reactnative.dev
- Expo Documentation: docs.expo.dev
- Expo Router Guide: expo.github.io/router
- AsyncStorage (React Native): react-native-async-storage/docs
- react-native-location (lib GPS usada): github.com/TimWei/react-native-location

- **Comunicación en Tiempo Real:**
- Socket.IO Documentation: socket.io/docs
- Eclipse Mosquitto (MQTT) Docs: mosquitto.org
- MQTT Protocol Basics: mqtt.org
- **Mapas y Geolocalización:**
- Google Maps Platform (Android) Setup: developers.google.com/maps/documentation/android-sdk
- Mapbox GL JS / React Native: docs.mapbox.com (según lo implementado)
- Geolocation API Expo: docs.expo.dev/versions/latest/sdk/location
- **Herramientas s DevOps:**
- GitHub Actions: docs.github.com/actions
- Docker: docs.docker.com
- Nginx (Reverse Proxy) Guide: nginx.com/resources/wiki/start/topics/examples/full

Estas referencias cubren las tecnologías empleadas, proporcionando información más profunda para ayudar en desarrollos futuros o resolución de dudas técnicas específicas.

15.2 Repositorio y Control de Versiones

- **Repositorio Git:** GitHub - AFSB114/UrbanTracker – Código fuente del proyecto (rama principal y dev).
- **Historial de Cambios:** Revisar la pestaña "Commits" y "Releases" en el repositorio para ver cambios específicos entre versiones. Los merges y mensajes de commit (por ejemplo, *"feat(shipment): add Shipment CRUD..."*) ofrecen contexto de funcionalidades añadidas[109].
- **Seguimiento de Issues:** Si se habilita GitHub Issues para este repo, allí se documentarán bugs conocidos y mejoras planificadas. En caso de proyectos privados, usar el sistema interno de la empresa para tickets de incidencia.

15.3 Información de Contacto

Para cualquier consulta, reporte de incidente o soporte técnico referente a UrbanTracker, a continuación, se proporciona la información de contacto del equipo responsable:

- **Equipo de Desarrollo:**

- Brayan Estiven Carvajal – Desarrollador Frontend
- Diego Fernando Cuellar – Desarrollador Backend
- Andrés Felipe Suaza - Desarrollador Fullstack
- Carlos Javier Rodríguez – Desarrollador Backend
  (Cada miembro se puede contactar vía el correo corporativo o GitHub del proyecto.)

[1] [2] [3] [4] [6] [15] [16] README.md

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/README.md

[5] page.tsx

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Web-Admin/src/app/page.tsx

[7] [52] [54] UserController.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/controller/Public/UserController.java

[8] [9] [13] [14] [46] [47] [53] [55] [56] UserService.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/service/UserService.java

[10] IUser.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/repository/IUser.java

[11] [65] [66] [67] [68] [69] [70] [92] User.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/model/User.java

[12] [36] [40] [41] [42] [43] authService.ts

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Movil-Driver-Client/src/services/api/authService.ts

[17] App.tsx

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Movil-User-Client/App.tsx

[18] [33] [99] [100] [104] app.json

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Movil-Driver-Client/app.json

[19] [37] [38] [39] JwtAuthenticationFilter.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/JWT/JwtAuthenticationFilter.java

[20] [21] Sidebar.tsx

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Web-Client/src/components/Sidebar.tsx

[22] [51] [95] [96] [97] [98] [105] [106] [107] [108] TROUBLESHOOT_LOCATION.md

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Movil-Driver-Client/TROUBLESHOOT_LOCATION.md

[23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [103] pom.xml

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/pom.xml

[34] [35] [49] JwtService.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/service/JWT/JwtService.java

[44] [71] VehicleStatus.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/model/VehicleStatus.java

[45] [58] [59] [60] RoleService.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/service/RoleService.java

[48] ApplicationConfig.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/config/ApplicationConfig.java

[50] [91] [94] CompanyService.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/service/CompanyService.java

[57] RoleController.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/controller/Private/RoleController.java

[61] VehicleController.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/controller/Private/VehicleController.java

[62] RouteController.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/controller/Private/RouteController.java

[63] RoutePointController.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/controller/Private/RoutePointController.java

[64] JourneyController.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/controller/Private/JourneyController.java

[72] [73] [74] [75] [76] [77] [78] [79] [80] [81] Company.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/model/Company.java

[82] [83] [84] [85] [86] [87] Vehicle.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/model/Vehicle.java

[88] [89] VehicleStatus.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/enums/VehicleStatus.java

[90] JourneyService.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/main/java/com/sena/urbantracker/service/JourneyService.java

[93] UrbanTrackerApplicationTests.java

https://github.com/AFSB114/UrbanTracker/blob/4e264ccb48444b0dcc8ba8ff80ee6c093d03bb2d/Backend/src/test/java/com/sena/urbantracker/UrbanTrackerApplicationTests.java

[101] Merge(merge): integrate changes from main into dev

https://github.com/AFSB114/UrbanTracker/pull/6

[102] chore(main): merge branch 'dev' into main

https://github.com/AFSB114/UrbanTracker/pull/5

[109] feat(shipment): add Shipment CRUD with JPA repository, service, and c…

https://github.com/code-sena/laboratory-2899747/pull/46