

Documentación Técnica de UpTrackAI

Equipo UpTrackAI

2026-02-02

Table of contents

1	Bienvenido a UpTrackAI	5
1.1	Objetivo del Sistema	5
1.2	Características Principales	5
1.3	Stack Tecnológico	6
1.4	Arquitectura de Alto Nivel	6
1.5	Estructura de la Documentación	6
1.6	Requisitos del Sistema	6
1.6.1	Desarrollo Local	6
1.6.2	Variables de Entorno Esenciales	7
1.7	Inicio Rápido	7
I	Arquitectura del Sistema	9
2	Arquitectura del Sistema	10
2.1	Resumen Ejecutivo	10
2.2	Capas de la Arquitectura	10
2.3	Módulos del Sistema	11
2.3.1	Inicialización de Módulos	11
2.4	Componentes Clave	11
2.4.1	Backend (Go)	11
2.4.2	Frontend (React/TypeScript)	11
2.4.3	Base de Datos (PostgreSQL)	11
2.5	Flujos Principales	12
2.5.1	1. Flujo de Health Check	12
2.5.2	2. Flujo de Autenticación	12
2.5.3	3. Flujo de Vinculación Telegram	12
2.6	Documentación Detallada	12
3	Visión General del Sistema	13
3.1	Diagrama de Contexto (C4 Nivel 1)	13
3.1.1	Actores y Sistemas	14
3.2	Alcance Funcional	15
3.3	Clean Architecture	15
3.4	Patrones de Diseño Implementados	16
3.4.1	Patrón Repository	16
3.4.2	Patrón Worker Pool	16
3.4.3	Patrón Module (Composición de Dependencias)	16
3.5	Modelo de Concurrencia	16
3.6	Gestión de Estados	17
3.7	Variables de Entorno	17
4	Backend y Servicios	18
4.1	Arquitectura de Componentes (C4 Nivel 3)	18

4.2	Arquitectura de Módulos	19
4.2.1	Estructura de Directorios	19
4.3	Módulo de Monitoreo	20
4.3.1	Entidades de Dominio	20
4.3.2	Value Objects	21
4.3.3	Scheduler y Orquestador	21
4.3.4	Algoritmo de Detección de Estado	23
4.4	Módulo de Notificaciones	23
4.4.1	Sistema de Alertas	23
4.4.2	Magic Link para Telegram	24
4.5	Módulo de Seguridad	24
4.5.1	Autenticación JWT	24
4.5.2	Seguridad de Contraseñas	24
4.6	API REST	25
4.6.1	Estándar de Respuestas	25
4.6.2	Endpoints Principales	25
4.6.3	Documentación Swagger	26
4.7	Stack Tecnológico	26
4.8	Ejecución Local	26
4.8.1	Requisitos	26
4.8.2	Comandos	26
4.8.3	Configuración Mínima (.env)	27
5	Frontend y Experiencia de Usuario	28
5.1	Diagrama de Contenedores (C4 Nivel 2)	28
5.2	Arquitectura de la Aplicación	29
5.3	Estructura del Proyecto	29
5.4	Sistema de Rutas	29
5.5	Cliente API	30
5.6	Páginas Principales	31
5.6.1	Dashboard	31
5.6.2	TargetDetail	31
5.6.3	Profile	31
5.7	Componentes UI	31
5.7.1	Sistema de Diseño	31
5.7.2	Componentes Reutilizables	32
5.7.3	Visualización de Estado	32
5.8	Gráficos con Recharts	32
5.9	Integración con Telegram	33
5.10	Stack Tecnológico	33
5.11	Desarrollo Local	33
5.11.1	Requisitos	33
5.11.2	Instalación y Ejecución	33
5.11.3	Scripts Disponibles	34
5.12	Despliegue con Docker	34
5.12.1	Dockerfile Multi-etapa	34
5.12.2	Configuración Nginx para SPA	35
5.12.3	Comandos Docker	35
5.13	Accesibilidad	35
6	Aplicación Móvil	36
6.1	Características Principales	36

6.2	Stack Tecnológico	36
6.3	Estructura del Proyecto	36
6.4	Sistema de Navegación	37
6.4.1	Pantallas Disponibles	38
6.5	Cliente API	38
6.6	Contexto de Autenticación	39
6.7	Widget Android	40
6.7.1	Características del Widget	40
6.8	Tema y Colores	41
6.9	Configuración de Expo	41
6.10	Desarrollo Local	42
6.10.1	Requisitos	42
6.10.2	Instalación y Ejecución	42
6.10.3	Configuración del API	42
6.11	Build de Producción	43
6.11.1	EAS Build (Recomendado)	43
6.11.2	Configuración EAS (eas.json)	43
6.12	Funcionalidades Futuras	43
7	Modelo de Datos	45
7.1	Diagrama de Clases / Entidades	45
7.2	Tablas del Sistema	45
7.2.1	users	45
7.2.2	credentials	46
7.2.3	monitoring_targets	46
7.2.4	check_results	47
7.2.5	metrics	47
7.2.6	target_statistics	47
7.2.7	notification_channels	48
7.2.8	telegram_linking_tokens	48
7.3	Migraciones	48
7.4	Consideraciones de Diseño	49
7.4.1	Separación de Datos Calientes/Fríos	49
7.4.2	Estrategia de Índices	49
7.4.3	UUID v7 como Primary Key	50
7.5	Conexión y Pool	50
7.6	Backup y Recuperación	51
7.6.1	Estrategia Recomendada	51
7.6.2	Comandos de Backup	51
7.7	Limpieza de Datos	51

1 Bienvenido a UpTrackAI

UpTrackAI es una plataforma SaaS de monitoreo de disponibilidad diseñada para verificar el estado de APIs, sitios web y servicios en tiempo real. Esta documentación técnica detalla la arquitectura, componentes y flujos de trabajo del sistema.

1.1 Objetivo del Sistema

Proporcionar a desarrolladores y administradores de sistemas una herramienta robusta para:

- **Monitorear disponibilidad:** Verificar que servicios críticos estén accesibles mediante health checks HTTP/TCP periódicos
- **Detectar anomalías:** Identificar patrones de comportamiento como degradación, inestabilidad o caídas
- **Alertar en tiempo real:** Notificar cambios de estado a través de Telegram (con soporte futuro para Email, Slack, etc.)
- **Analizar métricas:** Visualizar historial de latencia, uptime y tendencias de rendimiento
- **Gestionar equipos:** Permitir que múltiples usuarios administren sus propios objetivos de monitoreo

1.2 Características Principales

Característica	Descripción
Multi-tenant	Cada usuario gestiona sus propios targets de forma aislada
Detección inteligente	6 estados posibles (UP, DOWN, DEGRADED, UNSTABLE, FLAPPING, UNKNOWN)
Alta frecuencia	Verificaciones configurables desde 1 minuto hasta 24 horas
Notificaciones instantáneas	Integración con Telegram mediante Magic Link sin fricción
Dashboard interactivo	Visualización de métricas con gráficos de latencia y heatmaps
API REST documentada	Swagger/OpenAPI para integraciones externas

1.3 Stack Tecnológico

FRONTEND				
React 19	TypeScript	Vite	Tailwind CSS	Recharts
BACKEND				
Go 1.21+	Gin	GORM	JWT	Worker Pool
PERSISTENCIA				
PostgreSQL 15+	UUID v7	Índices optimizados		
INTEGRACIONES				
Telegram Bot API	HTTP/TCP Health Checks			

1.4 Arquitectura de Alto Nivel

El sistema sigue Clean Architecture con separación clara de responsabilidades:

- **Presentación:** API REST (Gin), Handlers HTTP, Respuestas JSON estandarizadas
- **Aplicación:** Casos de uso, DTOs, Commands/Queries (CQRS simplificado)
- **Dominio:** Entidades, Value Objects, Reglas de negocio, Interfaces de repositorio
- **Infraestructura:** PostgreSQL (GORM), Telegram API, JWT, HTTP Client

1.5 Estructura de la Documentación

La documentación está organizada en las siguientes secciones:

Sección	Contenido
Visión General	Contexto del sistema, patrones de diseño, modelo de estados
Backend	Módulos Go, Scheduler, API REST, autenticación
Frontend Web	React, componentes, integración API, despliegue
App Móvil	React Native, Expo, Widget Android, navegación
Base de Datos	Modelo de datos, esquema SQL, índices, migraciones

1.6 Requisitos del Sistema

1.6.1 Desarrollo Local

Componente	Versión Mínima
Go	1.21+
Node.js	18+

Componente	Versión Mínima
PostgreSQL	15+
pnpm	8+ (recomendado)

1.6.2 Variables de Entorno Esenciales

```
# Base de datos
DB_HOST=localhost
DB_PORT=5432
DB_USER=postgres
DB_PASSWORD=password
DB_NAME=uptrackai_db

# Servidor
PORT=8080
GIN_MODE=debug

# Telegram (opcional)
TELEGRAM_BOT_TOKEN=your_bot_token
TELEGRAM_BOT_NAME=Uptrackapp_bot
```

1.7 Inicio Rápido

```
# 1. Clonar el repositorio
git clone https://github.com/your-org/uptrack.git
cd uptrack

# 2. Iniciar base de datos (Docker)
docker run -d --name postgres \
  -e POSTGRES_PASSWORD=password \
  -e POSTGRES_DB=uptrackai_db \
  -p 5432:5432 postgres:15

# 3. Iniciar backend
cd backend
cp .env.example .env
go run .

# 4. Iniciar frontend (en otra terminal)
cd uptrack-gui
pnpm install
echo "VITE_API_BASE_URL=http://localhost:8080" > .env
pnpm dev
```

Acceder a: - **Frontend:** <http://localhost:5173> - **API:** <http://localhost:8080> - **Swagger:** <http://localhost:8080/swagger/index.html>

Comience explorando la [Visión General](#) para entender la arquitectura del sistema.

Part I

Arquitectura del Sistema

2 Arquitectura del Sistema

Esta sección describe en detalle la arquitectura técnica de UpTrackAI, desde la visión de alto nivel hasta los detalles de implementación de cada componente.

2.1 Resumen Ejecutivo

UpTrackAI es un sistema de monitoreo de disponibilidad construido siguiendo los principios de **Clean Architecture**, lo que garantiza:

- **Testabilidad:** Cada capa puede probarse de forma aislada
- **Mantenibilidad:** Cambios en infraestructura no afectan lógica de negocio
- **Extensibilidad:** Nuevos canales de notificación o tipos de checks sin modificar el core

2.2 Capas de la Arquitectura

PRESENTACIÓN

- HTTP Handlers (Gin)
- Request/Response DTOs
- Middleware (Auth, CORS, Logging)
- Swagger Documentation

APLICACIÓN

- Casos de Uso (Commands/Queries)
- Application Services
- DTOs de transferencia
- Orquestación de dominio

DOMINIO

- Entidades (MonitoringTarget, User, etc.)
- Value Objects (TargetId, Email, etc.)
- Interfaces de Repositorio
- Reglas de negocio y validaciones

INFRAESTRUCTURA

- PostgreSQL Repositories (GORM)
- Telegram API Client
- JWT Token Service
- HTTP Health Checker

2.3 Módulos del Sistema

El backend está organizado en módulos independientes, cada uno con su propia estructura Clean Architecture:

Módulo	Responsabilidad	Dependencias
Monitoring	Gestión de targets, ejecución de health checks, métricas	Notifications (alertas)
Notifications	Envío de alertas, canales, vinculación Telegram	-
Security	Autenticación JWT, hashing de contraseñas	User
User	Perfil de usuario, preferencias	-

2.3.1 Inicialización de Módulos

```
// main.go - Composición de dependencias
notificationsModule := notifications.NewModule(db)
monitoringModule := monitoring.NewModule(db, notificationsModule.Service)
securityModule := security.NewModule(db)
userModule := user.NewModule(db)
```

2.4 Componentes Clave

2.4.1 Backend (Go)

- **API REST (Gin):** Punto de entrada para clientes Web y Móviles
- **Scheduler (Worker Pool):** Orquestador que gestiona la ejecución periódica de health checks
- **Monitoring Module:** Núcleo de lógica de negocio que ejecuta las verificaciones
- **Notification Module:** Gestiona canales y envío de alertas asíncronas
- **Security Module:** Autenticación JWT y autorización basada en roles

2.4.2 Frontend (React/TypeScript)

- **Dashboard:** Vista principal con estadísticas y alertas activas
- **Systems:** Lista completa de targets monitoreados
- **Target Detail:** Métricas detalladas, gráficos de latencia, heatmap
- **Profile:** Configuración de usuario y vinculación con Telegram

2.4.3 Base de Datos (PostgreSQL)

- **Datos de Monitoreo:** monitoring_targets, check_results, metrics, target_statistics
- **Datos de Usuario:** users, credentials
- **Datos de Notificaciones:** notification_channels, telegram_linking_tokens

2.5 Flujos Principales

2.5.1 1. Flujo de Health Check

Scheduler (10s) → GetDueTargets → WorkerPool → HealthChecker
→ ResultAnalyzer → StateUpdater → NotificationDispatcher

2.5.2 2. Flujo de Autenticación

Login Request → Validate Credentials → Generate JWT
→ Return Token → Client stores in localStorage

2.5.3 3. Flujo de Vinculación Telegram

User clicks "Connect" → Generate Magic Link Token
→ Redirect to Telegram → Bot receives /start TOKEN
→ Backend validates → Associate ChatID with User

2.6 Documentación Detallada

Continúe con las siguientes secciones para información específica:

- [Visión General](#) - Contexto, patrones de diseño, modelo de estados
- [Backend](#) - Módulos Go, API REST, Scheduler, autenticación
- [Frontend](#) - React, componentes, integración, despliegue
- [Base de Datos](#) - Modelo de datos, esquema, índices, migraciones

3 Visión General del Sistema

UpTrackAI es una plataforma SaaS de monitoreo de disponibilidad diseñada para verificar el estado de APIs, sitios web y servicios en tiempo real. El sistema detecta caídas, degradaciones y comportamientos inestables, notificando a los usuarios a través de múltiples canales.

3.1 Diagrama de Contexto (C4 Nivel 1)

El siguiente diagrama muestra cómo UpTrackAI interactúa con sus usuarios y sistemas externos.

Tipo de Diagrama: Contexto C4
Sistema: UpTrackAI
Versión: 1.0
Fecha: 05/01/2026

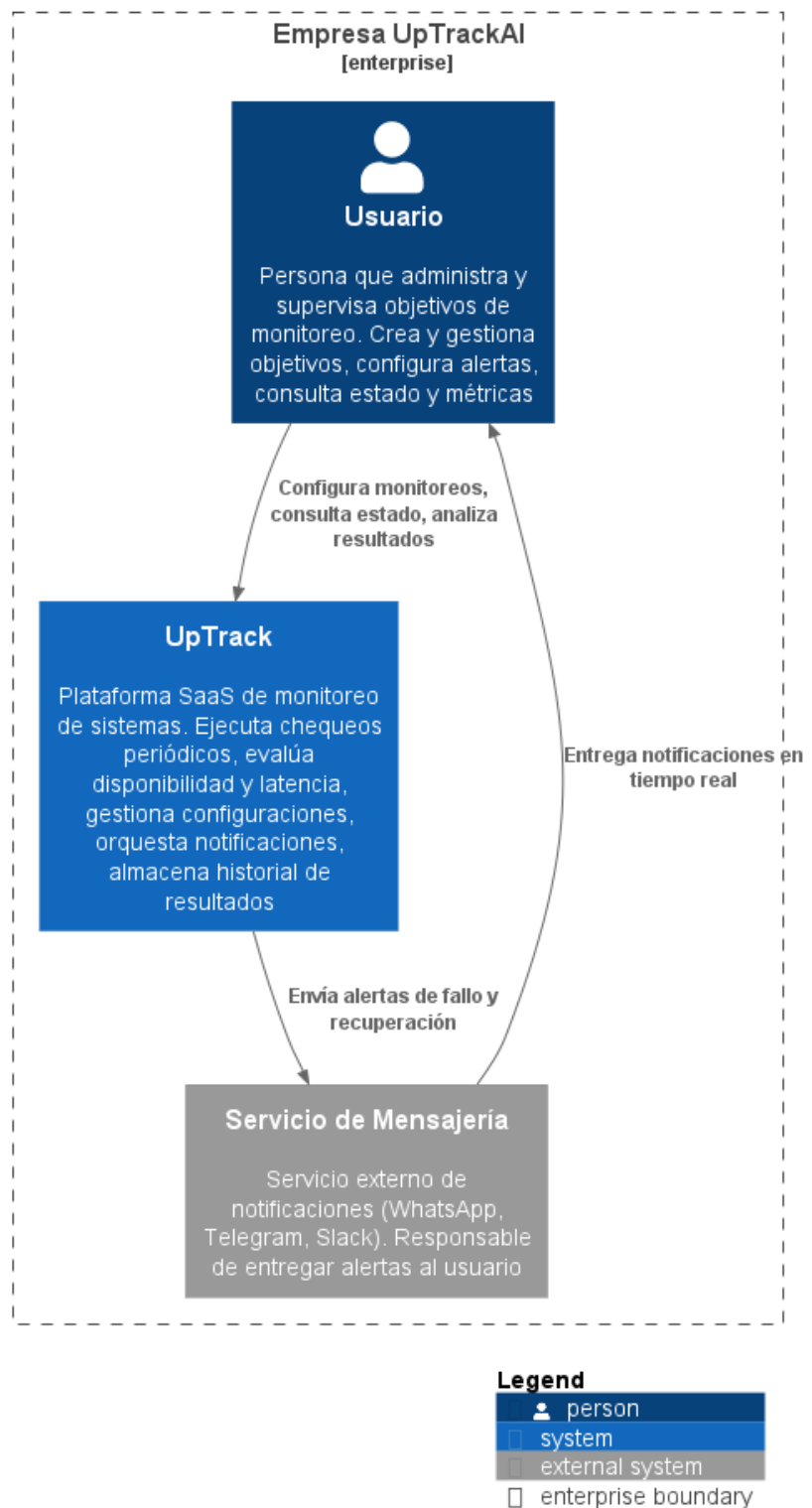


Figure 3.1: Diagrama de Contexto

3.1.1 Actores y Sistemas

Actor/Sistema	Descripción	Interacción
Usuario Final	Desarrollador o administrador de sistemas que monitorea sus servicios	Configura targets, recibe alertas, consulta métricas
Administrador	Usuario con permisos elevados para gestión global	Acceso a todos los targets, gestión de usuarios
APIs/Sitios Monitoreados	Endpoints externos que son verificados periódicamente	Reciben requests HTTP HEAD/GET cada N segundos
Telegram	Plataforma de mensajería para notificaciones	Recibe alertas de cambio de estado
PostgreSQL	Base de datos relacional	Almacena targets, resultados, métricas y usuarios

3.2 Alcance Funcional

El sistema cubre el ciclo completo de monitoreo:

1. **Configuración:** El usuario registra URLs a monitorear (APIs REST, sitios web)
2. **Verificación:** El scheduler ejecuta health checks periódicos
3. **Análisis:** El sistema detecta patrones de comportamiento (estable, degradado, flapping)
4. **Notificación:** Alertas en tiempo real cuando el estado cambia
5. **Visualización:** Dashboard con métricas históricas y estadísticas

3.3 Clean Architecture

UpTrackAI implementa Clean Architecture con separación estricta de responsabilidades:

```

CAPA DE PRESENTACIÓN
HTTP Handlers (Gin)  WebSocket  CLI

CAPA DE APLICACIÓN
Casos de Uso  DTOs  Commands/Queries

CAPA DE DOMINIO
Entidades  Value Objects  Reglas de Negocio  Interfaces

CAPA DE INFRAESTRUCTURA
PostgreSQL (GORM)  Telegram API  JWT  HTTP Client

```

Principios aplicados:

- **Dependency Inversion:** Las capas internas definen interfaces, las externas las implementan
- **Single Responsibility:** Cada módulo tiene una única razón de cambio
- **Open/Closed:** Extensible sin modificar código existente (nuevos senders de notificación)

3.4 Patrones de Diseño Implementados

3.4.1 Patrón Repository

Abstrae el acceso a datos permitiendo intercambiar implementaciones:

```
// Interfaz en dominio (sin dependencias externas)
type MonitoringTargetRepository interface {
    Save(target *MonitoringTarget) (*MonitoringTarget, error)
    GetByID(id TargetId) (*MonitoringTarget, error)
    GetDueTargets() ([]*MonitoringTarget, error)
    Delete(id TargetId) error
}

// Implementación en infraestructura (conoce GORM)
type PostgresMonitoringTargetRepository struct {
    db *gorm.DB
}
```

3.4.2 Patrón Worker Pool

Procesa múltiples targets concurrentemente con control de recursos:

```
type WorkerPool struct {
    workers      int
    jobQueue     chan *MonitoringTarget
    wg           sync.WaitGroup
    processor    func(*MonitoringTarget)
}
```

3.4.3 Patrón Module (Composición de Dependencias)

Cada dominio se inicializa como un módulo independiente:

```
// Cada módulo encapsula sus capas
monitoringModule := monitoring.NewModule(db, notificationService)
securityModule  := security.NewModule(db)
userModule      := user.NewModule(db)
```

3.5 Modelo de Concurrencia

El sistema utiliza goroutines y channels de Go para operaciones paralelas:

Componente	Tipo	Descripción
Worker Pool	Goroutines	4 workers procesando targets en paralelo

Componente	Tipo	Descripción
Notification Dispatcher	Channel (buffer 100)	Cola asíncrona de alertas
Telegram Poller	Goroutine	Escucha mensajes del bot en background
HTTP Server	Goroutine	Servidor web en thread separado

3.6 Gestión de Estados

El sistema detecta 6 estados posibles para cada target monitoreado:

Estado	Descripción	Criterio de Detección
UP	Servicio saludable	3 checks consecutivos exitosos en 4 intentos
DOWN	Servicio inalcanzable	3 checks consecutivos fallidos
DEGRADED	Respuesta lenta	Latencia > umbral configurado
UNSTABLE	Fallos intermitentes	3 iguales en 5-9 intentos
FLAPPING	Cambios rápidos	No se logran 3 iguales en 12 intentos
UNKNOWN	Sin datos suficientes	Estado inicial o error de medición

3.7 Variables de Entorno

El sistema se configura mediante variables de entorno:

Variable	Descripción	Valor por Defecto
DB_HOST	Host de PostgreSQL	localhost
DB_PORT	Puerto de PostgreSQL	5432
DB_USER	Usuario de base de datos	postgres
DB_PASSWORD	Contraseña de base de datos	-
DB_NAME	Nombre de la base de datos	uptrackai_db
PORT	Puerto del servidor HTTP	8080
GIN_MODE	Modo de Gin (debug/release)	debug
TELEGRAM_BOT_TOKEN	Token del bot de Telegram	-
TELEGRAM_BOT_NAME	Nombre del bot	Uptrackapp_bot
SKIP_CONNECTIVITY_CHECK	Omitir verificación de red	false
SECURITY_DISABLED	Desactivar autenticación (dev)	false

4 Backend y Servicios

El backend de UpTrackAI está construido en Go 1.21+, aprovechando su modelo de concurrencia nativo para ejecutar health checks en paralelo de manera eficiente.

4.1 Arquitectura de Componentes (C4 Nivel 3)

El siguiente diagrama detalla los módulos internos del Backend API y sus interacciones.

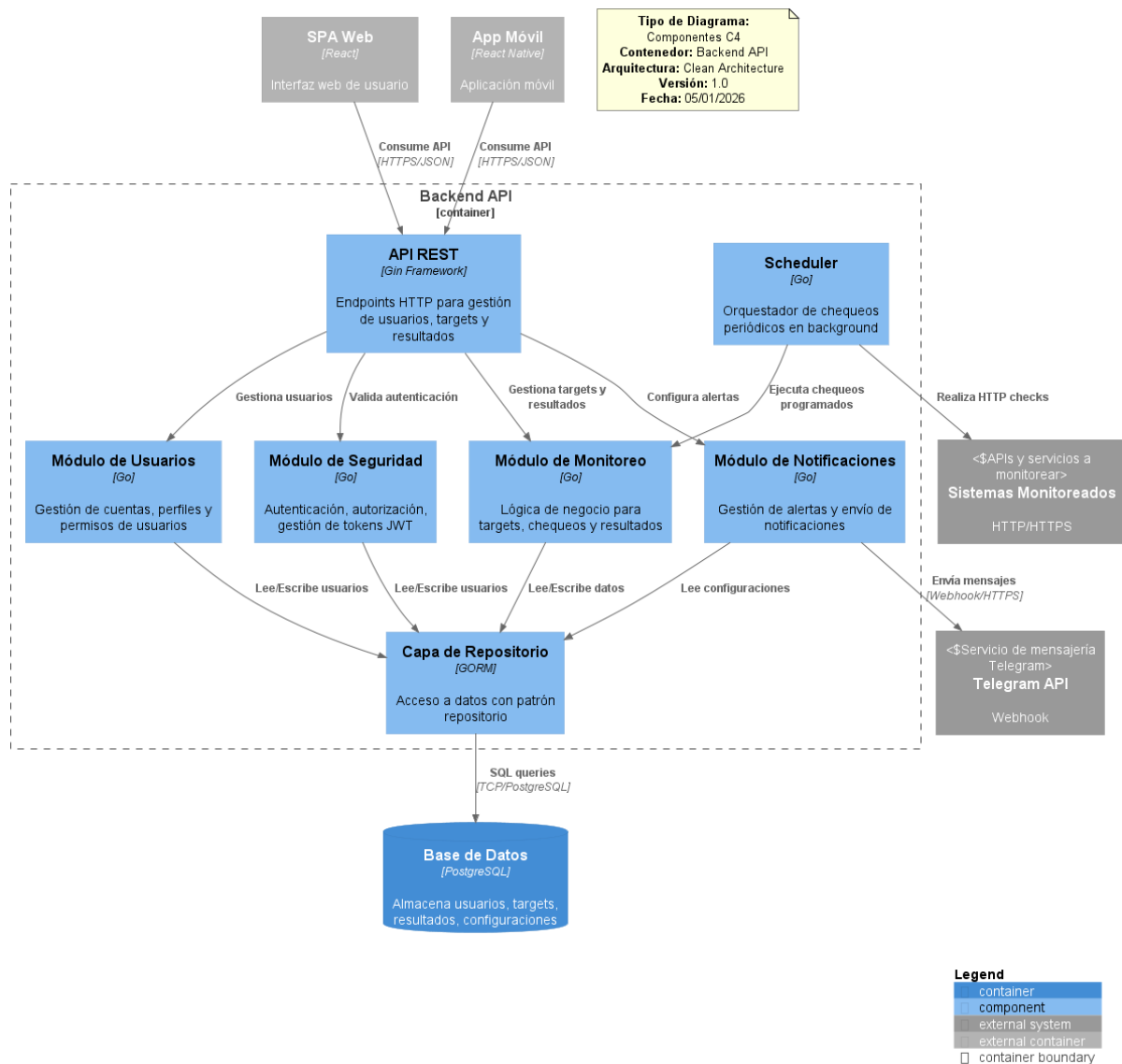


Figure 4.1: Diagrama de Componentes

4.2 Arquitectura de Módulos

El sistema está organizado en módulos independientes, cada uno siguiendo Clean Architecture internamente.

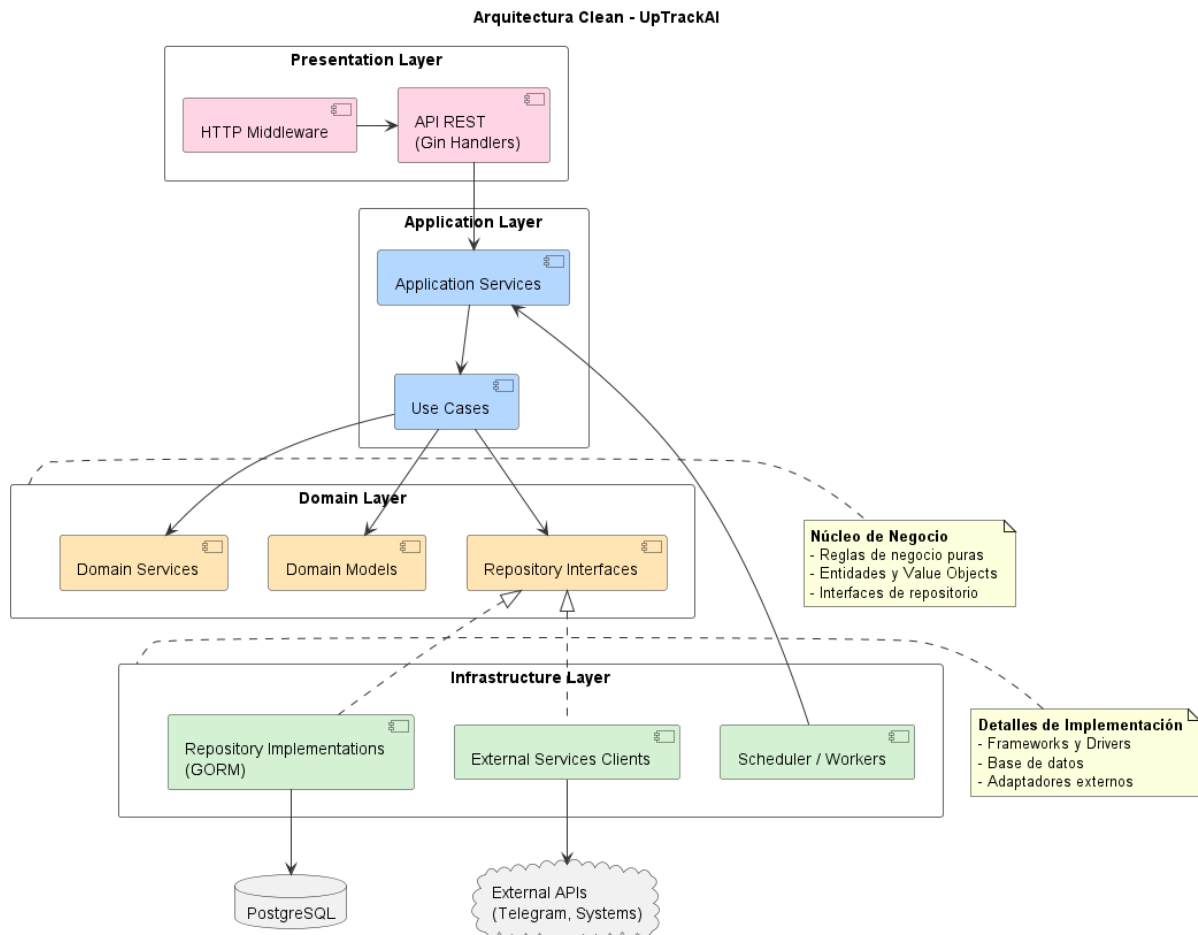


Figure 4.2: Arquitectura Clean

4.2.1 Estructura de Directorios

```
backend/
  main.go           # Punto de entrada
  config/           # Configuración de infraestructura
    database.go     # Conexión PostgreSQL
    migrations.go   # Auto-migraciones GORM
    server.go       # Configuración HTTP
    telemetry.go    # Observabilidad
  internal/
    app/            # Utilidades compartidas (APIResponse)
    monitoring/     # Módulo de monitoreo
      domain/       # Entidades y reglas de negocio
      application/  # Casos de uso y DTOs
      infrastructure/ # Repositorios PostgreSQL
      presentation/ # Handlers HTTP
```

```

    scheduler/      # Orquestador de health checks
notifications/    # Módulo de notificaciones
    domain/        # AlertEvent, Channels, Senders
    application/   # NotificationService, Linking
infrastructure/   # Telegram API, Repos
presentation/     # Webhooks y handlers
security/         # Autenticación y autorización
    domain/        # Credentials, Tokens
    application/   # AuthService
infrastructure/   # JWT, Bcrypt
presentation/     # Login/Register handlers
user/            # Gestión de usuarios
server/          # Middleware compartido
docs/            # Swagger generado

```

4.3 Módulo de Monitoreo

4.3.1 Entidades de Dominio

MonitoringTarget - Representa un servicio a monitorear:

```

type MonitoringTarget struct {
    userId      UserId
    targetId    TargetId
    name        string
    url         string
    isActive    bool
    previousStatus TargetStatus // Para detectar cambios
    currentStatus TargetStatus
    createdAt   time.Time
    lastCheckedAt time.Time
    lastResponseTime int           // ms
    targetType   TargetType   // API | WEB
    configuration *CheckConfiguration
}

```

CheckResult - Resultado de una verificación individual:

```

type CheckResult struct {
    id          CheckResultId
    targetId    TargetId
    timestamp   time.Time
    responseTimeMs int
    reachable   bool
    status      TargetStatus
    errorMessage string
}

```

TargetStatistics - Métricas agregadas históricas:

```

type TargetStatistics struct {
    targetId      TargetId
    avgResponseTimeMs int
    totalChecksCount int
    lastUpdatedAt  time.Time
}

```

4.3.2 Value Objects

Value Object	Tipo	Descripción
TargetId	string	UUID v7 del target
TargetStatus	enum	UP, DOWN, DEGRADED, FLAPPING, UNSTABLE, UNKNOWN
TargetType	enum	API, WEB
CheckConfiguration	struct	Intervalo, timeout, umbral de latencia

4.3.3 Scheduler y Orquestador

El sistema de scheduling ejecuta health checks de manera periódica y concurrente.

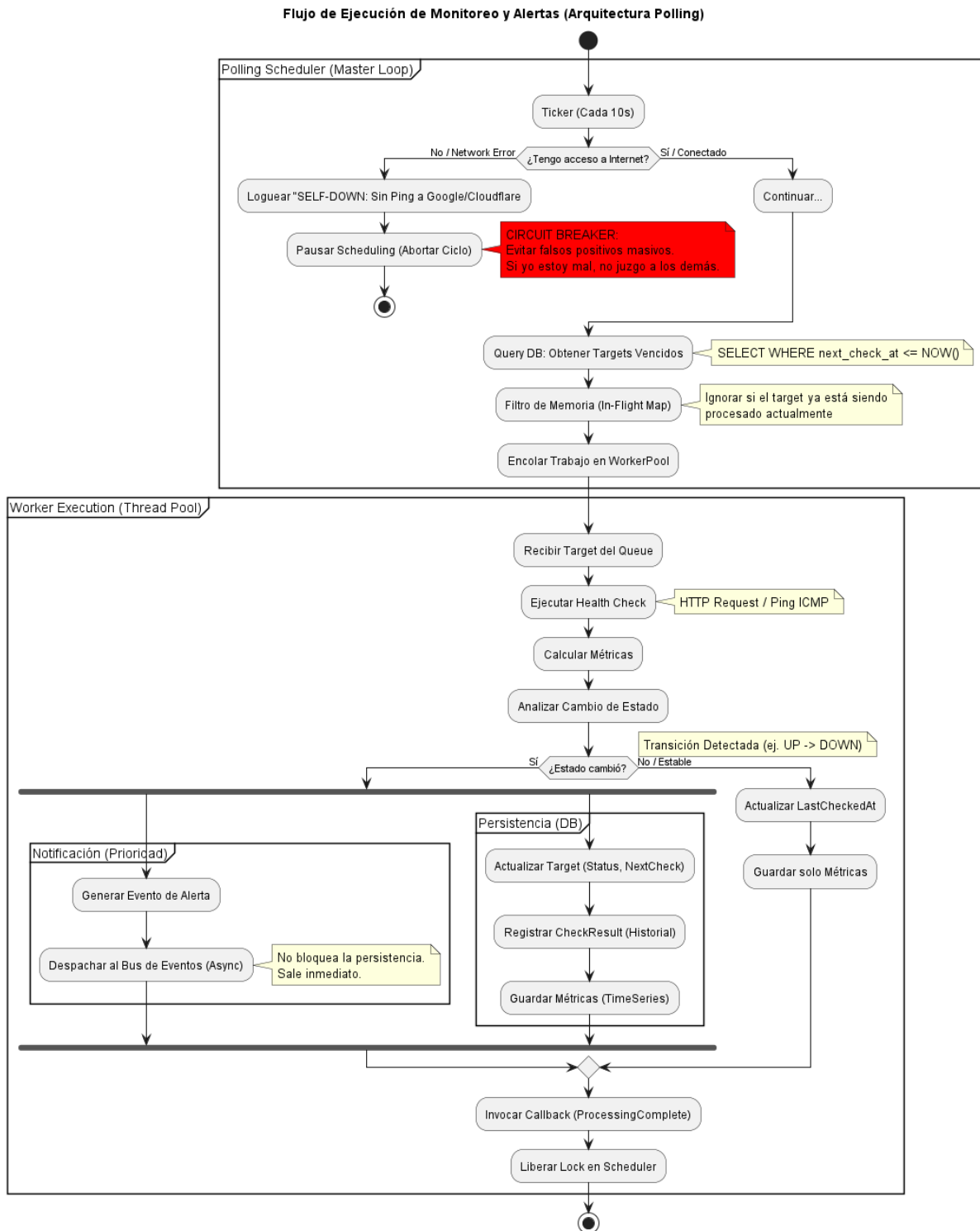


Figure 4.3: Flujo de Monitoreo

Componentes del Scheduler:

Componente	Responsabilidad
PollingScheduler	Loop principal cada 10s, coordina ejecución
Orchestrator	Orquesta el pipeline completo de un target

Componente	Responsabilidad
WorkerPool	Pool de 4 goroutines procesando en paralelo
HealthChecker	Ejecuta HTTP requests y determina estado
ResultAnalyzer	Analiza sesión de pings para determinar estado final
StateUpdater	Persiste cambios en base de datos
NotificationDispatcher	Cola asíncrona de alertas (buffer 100)

4.3.4 Algoritmo de Detección de Estado

El sistema realiza hasta 12 pings por sesión para determinar el estado:

1. Ejecutar ping HTTP HEAD/GET
2. Repetir hasta conseguir 3 resultados consecutivos iguales
3. Si se logra en 4 intentos → Estado estable (UP/DOWN)
4. Si se logra en 5-9 intentos → UNSTABLE
5. Si no se logra en 12 intentos → FLAPPING

4.4 Módulo de Notificaciones

4.4.1 Sistema de Alertas

El módulo de notificaciones gestiona el envío de alertas a través de múltiples canales.

AlertEvent - Evento de alerta:

```
type AlertEvent struct {
    userId      string
    title       string
    message     string
    severity    Severity    // CRITICAL, WARNING, INFO, RESOLVED
    prevSeverity Severity
    source      string
    alertType   AlertType  // MONITORING, SYSTEM
    metadata    map[string]string
    timestamp   time.Time
}
```

ShouldNotify() determina si la alerta debe enviarse:

- Cambio de severidad (ej: WARNING → CRITICAL)
- Nueva alerta CRITICAL o WARNING
- Resolución de un problema previo

4.4.2 Magic Link para Telegram

El sistema implementa vinculación sin fricción de cuentas de Telegram:

1. Usuario hace click en “Conectar Telegram” en el Frontend
2. Backend genera un token único con TTL de 15 minutos
3. Frontend redirige al usuario a Telegram con deep link
4. Usuario envía `/start TOKEN` al bot
5. Bot valida el token y asocia el ChatID al usuario
6. Confirmación de vinculación exitosa

4.5 Módulo de Seguridad

4.5.1 Autenticación JWT

El sistema utiliza JWT (JSON Web Tokens) para autenticación stateless:

```
// Estructura del token
type Claims struct {
    UserID string `json:"user_id"`
    Email  string `json:"email"`
    Role   string `json:"role"`
    jwt.RegisteredClaims
}
```

Endpoints de Autenticación:

Método	Ruta	Descripción
POST	/api/v1/register	Registro de usuario
POST	/api/v1/login	Login, retorna JWT

Middleware de Extracción:

```
func ExtractUserID(tokenService TokenGenerator) gin.HandlerFunc {
    return func(c *gin.Context) {
        // Extrae token del header Authorization: Bearer <token>
        // Valida firma y expiración
        // Inyecta userID y role en el contexto
    }
}
```

4.5.2 Seguridad de Contraseñas

- Hashing con **bcrypt** (cost factor 10)
- Validación de longitud mínima (8 caracteres)
- No se almacenan contraseñas en texto plano

4.6 API REST

4.6.1 Estándar de Respuestas

Todas las respuestas siguen el formato `APIResponse`:

```
{
  "success": true,
  "message": "targets_retrieved",
  "data": [...],
  "links": {
    "self": "/api/v1/targets",
    "next": "/api/v1/targets?page=2"
  },
  "meta": {
    "page": 1,
    "limit": 20,
    "total": 45
  }
}
```

4.6.2 Endpoints Principales

Monitoreo:

Método	Ruta	Descripción
GET	/api/v1/targets	Listar targets del usuario
POST	/api/v1/targets	Crear nuevo target
GET	/api/v1/targets/:id	Detalle de un target
DELETE	/api/v1/targets/:id	Eliminar target
PATCH	/api/v1/targets/:id/toggle	Activar/desactivar
PUT	/api/v1/targets/:id/configuration	Actualizar config
GET	/api/v1/targets/:id/metrics	Métricas del target
GET	/api/v1/targets/:id/history	Historial de checks
GET	/api/v1/targets/:id/statistics	Estadísticas agregadas

Notificaciones:

Método	Ruta	Descripción
GET	/api/v1/notifications/channels	Canales configurados
POST	/api/v1/telegram/link	Iniciar vinculación Telegram
POST	/api/webhooks/telegram	Webhook para Telegram

Autenticación:

Método	Ruta	Descripción
POST	/api/v1/register	Registrar usuario

Método	Ruta	Descripción
POST	/api/v1/login	Obtener token JWT

4.6.3 Documentación Swagger

La API está documentada con Swagger/OpenAPI. Accesible en:

<http://localhost:8080/swagger/index.html>

4.7 Stack Tecnológico

Categoría	Tecnología	Versión
Lenguaje	Go	1.21+
Framework HTTP	Gin	1.9+
ORM	GORM	1.25+
Base de Datos	PostgreSQL	15+
Autenticación	JWT (golang-jwt)	v5
Hashing	bcrypt	-
Documentación	Swaggo	1.16+
Variables de entorno	godotenv	1.5+

4.8 Ejecución Local

4.8.1 Requisitos

- Go 1.21+
- PostgreSQL 15+
- Variables de entorno configuradas

4.8.2 Comandos

```
# Instalar dependencias
go mod download

# Ejecutar en modo desarrollo
go run .

# Compilar binario
go build -o uptrack-api .

# Ejecutar tests
go test ./...
```

```
# Generar documentación Swagger
swag init
```

4.8.3 Configuración Mínima (.env)

```
DB_HOST=localhost
DB_PORT=5432
DB_USER=postgres
DB_PASSWORD=password
DB_NAME=uptrackai_db
DB_SSLMODE=disable
PORT=8080
GIN_MODE=debug
```

5 Frontend y Experiencia de Usuario

UpTrackAI ofrece una interfaz web moderna construida con React 19 y TypeScript, diseñada para proporcionar visibilidad en tiempo real del estado de los sistemas monitoreados.

5.1 Diagrama de Contenedores (C4 Nivel 2)

El sistema expone interfaces para usuarios finales y administradores.

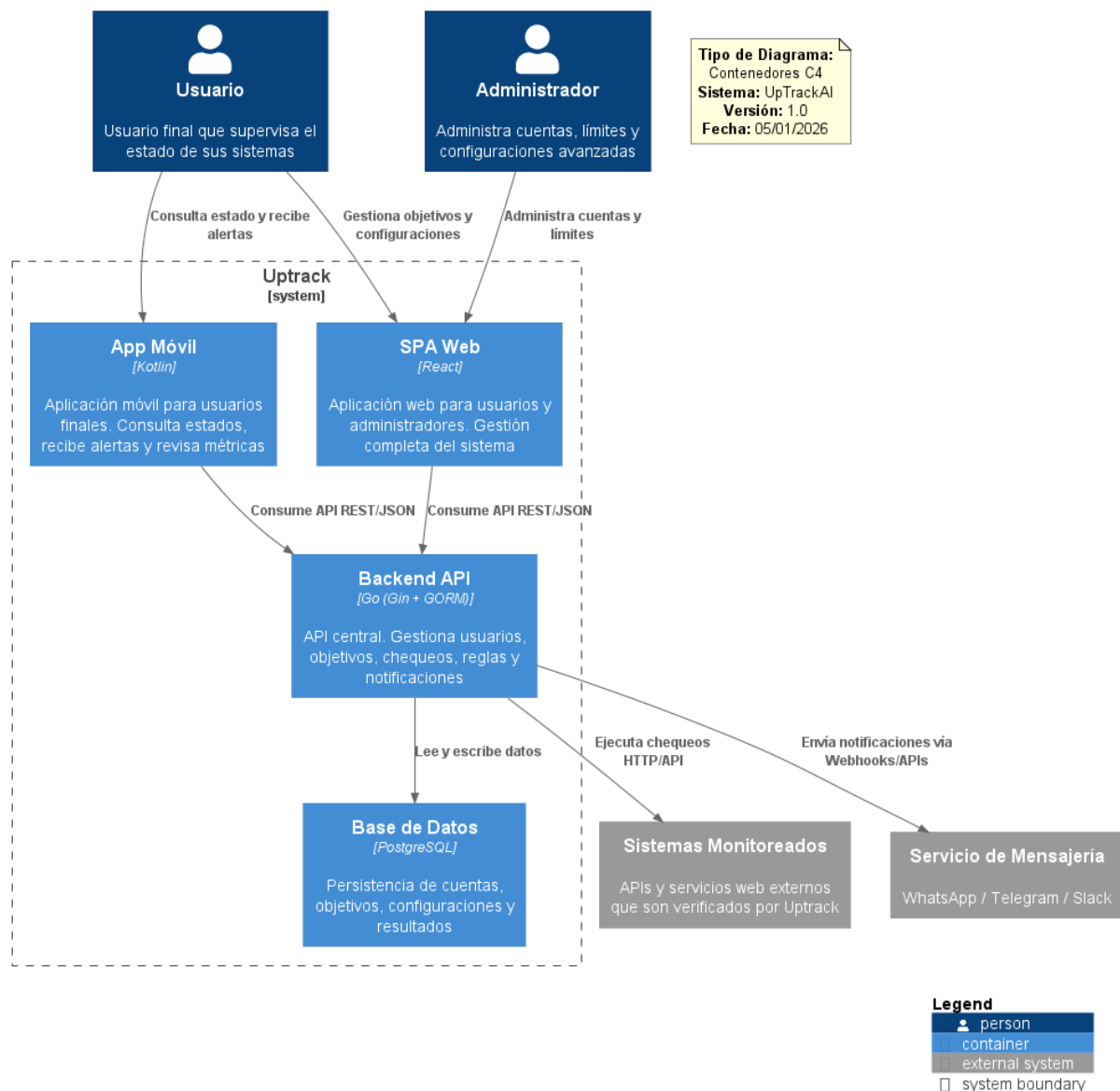


Figure 5.1: Diagrama de Contenedores

5.2 Arquitectura de la Aplicación

5.3 Estructura del Proyecto

```
uptrack-gui/
  index.html          # Punto de entrada HTML
  vite.config.ts      # Configuración de Vite
  tailwind.config.js  # Configuración de Tailwind CSS
  tsconfig.json       # Configuración TypeScript
  package.json        # Dependencias y scripts
  src/
    main.tsx          # Bootstrap de React
    App.tsx           # Router y rutas protegidas
    index.css          # Estilos globales y tema
    api/
      fetch.ts         # Cliente HTTP con auth
    components/
      layout/
        MainLayout.tsx # Layout principal
      ui/
        Button.tsx     # Botón reutilizable
        FormField.tsx  # Campo de formulario
        Modal.tsx      # Diálogo modal
        PageHeader.tsx # Cabecera de página
        NotificationBell.tsx # Campana de alertas
    context/
      ThemeContext.tsx # Modo claro/oscurο
    pages/
      Login.tsx        # Autenticación
      Dashboard.tsx    # Vista principal
      Systems.tsx      # Lista de targets
      TargetDetail.tsx # Detalle con métricas
      AddTarget.tsx    # Crear nuevo target
      Profile.tsx      # Perfil de usuario
      Settings.tsx     # Configuración
      Reports.tsx      # Reportes
    data/
      types.ts         # Definiciones TypeScript
```

5.4 Sistema de Rutas

La aplicación utiliza React Router v7 con rutas protegidas basadas en autenticación:

Ruta	Componente	Protegida	Descripción
/login	Login	No	Formulario de autenticación
/dashboard	Dashboard	Sí	Vista general con estadísticas
/systems	Systems	Sí	Lista completa de targets
/target/:id	TargetDetail	Sí	Métricas detalladas de un target
/add-target	AddTarget	Sí	Formulario para crear target

Ruta	Componente	Protegida	Descripción
/profile	Profile	Sí	Configuración de perfil y Telegram
/settings	Settings	Sí	Preferencias de la aplicación
/reports	Reports	Sí	Reportes y exportación

Protección de Rutas:

```
// App.tsx - Verificación de autenticación
const [isLoggedIn, setIsLoggedIn] = useState(!localStorage.getItem('token'));

<Route path="/dashboard" element={
  isLoggedIn ? (
    <MainLayout>
      <Dashboard />
    </MainLayout>
  ) : (
    <Navigate to="/login" />
  )
} />
```

5.5 Cliente API

El cliente HTTP centraliza la autenticación y manejo de sesión expirada:

```
// api/fetch.ts
export async function fetchWithAuth(url: string, options: RequestInit = {}) {
  const token = localStorage.getItem('token');

  const response = await fetch(`${import.meta.env.VITE_API_BASE_URL}${url}`, {
    ...options,
    headers: {
      ...options.headers,
      'Authorization': `Bearer ${token}`,
    },
  });

  // Redirección automática si el token expiró
  if (response.status === 401) {
    localStorage.removeItem('token');
    window.location.href = '/login';
    throw new Error('Unauthorized');
  }

  return response;
}
```

5.6 Páginas Principales

5.6.1 Dashboard

Vista principal que muestra:

- **Estadísticas globales:** Total de sistemas, online, con alertas
- **Lista de alertas:** Solo targets con problemas (DOWN, DEGRADED, FLAPPING, UNSTABLE)
- **Filtros:** Búsqueda por nombre/URL, filtro por estado

```
// Estructura de datos de un Target
interface Target {
  id: string;
  name: string;
  url: string;
  target_type: string;      // "API" | "WEB"
  current_status: string;   // "UP" | "DOWN" | ...
  last_checked_at: string;
  avg_response_time: number; // ms
}
```

5.6.2 TargetDetail

Vista detallada de un target con:

- **Información básica:** Nombre, URL, tipo, estado actual
- **Gráfico de latencia:** Histórico de tiempos de respuesta (Recharts)
- **Heatmap de disponibilidad:** Visualización por hora/día
- **Historial de checks:** Últimos resultados individuales
- **Configuración:** Intervalo, timeout, umbral de latencia

5.6.3 Profile

Gestión de perfil de usuario:

- Edición de nombre, idioma, zona horaria
- **Vinculación con Telegram:** Botón que genera Magic Link
- Visualización de canales de notificación activos

5.7 Componentes UI

5.7.1 Sistema de Diseño

La aplicación utiliza un sistema de tokens CSS para tematización:

```

/* index.css - Variables de tema */
:root {
  --background: #0a0a0a;
  --background-card: #111111;
  --background-input: #1a1a1a;
  --text-main: #ffffff;
  --text-muted: #a1a1aa;
  --primary: #3b82f6;
  --border-dark: #27272a;
  /* ... */
}

```

5.7.2 Componentes Reutilizables

Componente	Props	Descripción
Button	variant, size, loading, disabled	Botón con estados
FormField	label, error, type	Input con label y validación
Modal	isOpen, onClose, title	Diálogo modal
PageHeader	title, description, actions	Cabecera de página
NotificationBell	count	Indicador de alertas

5.7.3 Visualización de Estado

Códigos de color consistentes para estados:

```

const getStatusColor = (status: string) => {
  switch (status) {
    case 'UP':      return 'text-green-400'; // Verde
    case 'DOWN':    return 'text-red-400';   // Rojo
    case 'DEGRADED': return 'text-yellow-400'; // Amarillo
    case 'FLAPPING': return 'text-orange-400'; // Naranja
    case 'UNSTABLE': return 'text-purple-400'; // Púrpura
    default:        return 'text-gray-400';  // Gris
  }
};

```

5.8 Gráficos con Recharts

La aplicación utiliza Recharts para visualizaciones:

```

// Gráfico de latencia en TargetDetail
<ResponsiveContainer width="100%" height={300}>
  <AreaChart data={metricsData}>
    <CartesianGrid strokeDasharray="3 3" />
    <XAxis dataKey="timestamp" />
    <YAxis unit="ms" />

```



```

<Tooltip />
<Area
  type="monotone"
  dataKey="response_time"
  stroke="#3b82f6"
  fill="#3b82f680"
/>
</AreaChart>
</ResponsiveContainer>

```

5.9 Integración con Telegram

El flujo de vinculación desde el frontend:

1. Usuario navega a `/profile`
2. Click en “Conectar Telegram”
3. Frontend llama `POST /api/v1/telegram/link`
4. Backend retorna `{ deepLink: "t.me/bot?start=TOKEN" }`
5. Frontend abre el deep link (redirect a Telegram)
6. Usuario interactúa con el bot, vinculación completada

5.10 Stack Tecnológico

Categoría	Tecnología	Versión
Framework	React	19.2
Lenguaje	TypeScript	5.9
Build Tool	Vite	7.2
Estilos	Tailwind CSS	4.1
Routing	React Router	7.10
Gráficos	Recharts	3.5
Iconos	Lucide React	0.555
Linting	ESLint	9.39

5.11 Desarrollo Local

5.11.1 Requisitos

- Node.js 18+
- npnm (recomendado) o npm

5.11.2 Instalación y Ejecución

```
# Instalar dependencias
cd uptrack-gui
pnpm install

# Configurar variable de entorno
echo "VITE_API_BASE_URL=http://localhost:8080" > .env

# Ejecutar en desarrollo (HMR)
pnpm dev

# Compilar para producción
pnpm build

# Preview de producción
pnpm preview
```

5.11.3 Scripts Disponibles

Comando	Descripción
pnpm dev	Servidor de desarrollo con HMR (puerto 5173)
pnpm build	Compilación de producción
pnpm preview	Preview del build de producción
pnpm lint	Ejecutar ESLint

5.12 Despliegue con Docker

5.12.1 Dockerfile Multi-etapa

```
# Etapa 1: Construcción
FROM node:20-alpine AS builder
WORKDIR /app
COPY package.json pnpm-lock.yaml ./
RUN npm install -g pnpm && pnpm install --frozen-lockfile
COPY . .
ARG VITE_API_BASE_URL
ENV VITE_API_BASE_URL=$VITE_API_BASE_URL
RUN pnpm build

# Etapa 2: Servidor Web
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

5.12.2 Configuración Nginx para SPA

```
server {
    listen 80;
    server_name localhost;
    root /usr/share/nginx/html;
    index index.html;

    # Soporte para React Router (client-side routing)
    location / {
        try_files $uri $uri/ /index.html;
    }

    # Cache de assets estáticos
    location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2)$ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

5.12.3 Comandos Docker

```
# Construir imagen
docker build \
  --build-arg VITE_API_BASE_URL=https://api.uptrack.example.com \
  -t uptrack-gui .

# Ejecutar contenedor
docker run -d -p 3000:80 --name uptrack-frontend uptrack-gui
```

5.13 Accesibilidad

La aplicación implementa buenas prácticas de accesibilidad:

- **Roles ARIA:** `role="status"`, `role="alert"`, `aria-live`
- **Labels:** `aria-label` en elementos interactivos
- **Navegación por teclado:** `tabIndex` en elementos focalizables
- **Contraste:** Colores con ratio WCAG AA

```
// Ejemplo de accesibilidad en Dashboard
<article
  className="bg-background-card rounded-lg p-6"
  aria-label={` ${totalTargets} sistemas en total`}>
  <p className="text-3xl font-bold">{totalTargets}</p>
</article>
```

6 Aplicación Móvil

UpTrack Mobile es una aplicación nativa construida con React Native y Expo, diseñada para consulta rápida del estado de los sistemas monitoreados desde dispositivos Android e iOS.

6.1 Características Principales

Característica	Descripción
Dashboard	Vista general con estadísticas y alertas activas
Lista de Sistemas	Todos los targets con filtros por estado
Detalle de Target	Métricas, historial y configuración
Widget Android	Widget nativo para pantalla de inicio
Notificaciones Push	Alertas en tiempo real (futuro)
Modo Offline	Caché local con AsyncStorage

6.2 Stack Tecnológico

Categoría	Tecnología	Versión
Framework	React Native	0.81
Plataforma	Expo	54
Lenguaje	TypeScript	5.9
UI	React Native SVG	15.12
Storage	AsyncStorage	2.2
Widgets	react-native-android-widget	0.20
Runtime	React	19.1

6.3 Estructura del Proyecto

```
up-track-mobile/  
  App.tsx           # Punto de entrada  
  app.json          # Configuración Expo  
  eas.json          # Configuración EAS Build  
  index.ts          # Registro de la app  
  package.json       # Dependencias  
  tsconfig.json      # Configuración TypeScript  
  assets/           # Iconos y splash screens  
  src/  
    api/  
      config.ts      # URL base del API
```

```

    fetch.ts          # Cliente HTTP con auth
components/
  BottomTabBar.tsx
  Layout.tsx
context/
  AuthContext.tsx # Gestión de sesión
navigation/
  AppNavigator.tsx # Navegación entre pantallas
screens/
  LoginScreen.tsx
  DashboardScreen.tsx
  SystemsScreen.tsx
  TargetDetailsScreen.tsx
  AddTargetScreen.tsx
  SettingsScreen.tsx
  NotificationsScreen.tsx
  WidgetConfigScreen.tsx
theme/
  colors.ts          # Paleta de colores
widgets/
  SystemStatusWidget.tsx
  widget-task-handler.tsx

```

6.4 Sistema de Navegación

La aplicación utiliza un sistema de navegación basado en estado (sin React Navigation para mantener el bundle ligero):

```

// AppNavigator.tsx
type Screen = TabScreen | 'addTarget' | 'targetDetails' | 'notifications' |
  ↪ 'widgetConfig';

const AppNavigator: React.FC = () => {
  const { isLoggedIn, loading } = useAuth();
  const [currentScreen, setCurrentScreen] = useState<Screen>('dashboard');
  const [selectedTargetId, setSelectedTargetId] = useState<string |
  ↪ null>(null);

  // Navegación condicional basada en autenticación
  if (!isLoggedIn) {
    return <LoginScreen />;
  }

  // Renderizado de pantalla actual
  switch (currentScreen) {
    case 'dashboard':
      return <DashboardScreen onNavigateToDetails={navigateToDetails} />;
    case 'systems':
      return <SystemsScreen onNavigateToDetails={navigateToDetails} />;
    // ...

```

```
}  
};
```

6.4.1 Pantallas Disponibles

Pantalla	Tab	Descripción
LoginScreen	-	Autenticación de usuario
DashboardScreen		Estadísticas y alertas activas
SystemsScreen		Lista completa de targets
TargetDetailsScreen	-	Detalle con métricas de un target
AddTargetScreen	-	Formulario para crear target
SettingsScreen		Configuración de la app
NotificationsScreen	-	Preferencias de notificaciones
WidgetConfigScreen	-	Configuración del widget

6.5 Cliente API

El cliente HTTP centraliza la autenticación usando AsyncStorage:

```
// api/fetch.ts  
export async function fetchWithAuth(url: string, options: RequestInit = {}) {  
  const token = await AsyncStorage.getItem('token');  
  
  // Cache busting para peticiones GET  
  const method = options.method?.toUpperCase() || 'GET';  
  let finalUrl = url;  
  
  if (method === 'GET') {  
    const separator = url.includes('?') ? '&' : '?';  
    finalUrl = `${url}${separator}_t=${Date.now()}`;  
  }  
  
  const response = await fetch(`${API_BASE_URL}${finalUrl}`, {  
    ...options,  
    headers: {  
      ...options.headers,  
      'Authorization': `Bearer ${token}`,  
    },  
  });  
  
  // Logout automático si el token expiró  
  if (response.status === 401) {  
    await AsyncStorage.removeItem('token');  
    throw new Error('Unauthorized');  
  }  
  
  return response;  
}
```

```
}
```

6.6 Contexto de Autenticación

Gestión de sesión mediante React Context y AsyncStorage:

```
// context/AuthContext.tsx
interface AuthContextType {
  isLoggedIn: boolean;
  token: string | null;
  loading: boolean;
  login: (token: string) => Promise<void>;
  logout: () => Promise<void>;
}

export const AuthProvider: React.FC<{ children: ReactNode }> = ({ children })
  => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  const [token, setToken] = useState<string | null>(null);

  // Verificar token almacenado al iniciar
  useEffect(() => {
    const checkAuth = async () => {
      const storedToken = await AsyncStorage.getItem('token');
      if (storedToken) {
        setToken(storedToken);
        setIsLoggedIn(true);
      }
    };
    checkAuth();
  }, []);

  const login = async (newToken: string) => {
    await AsyncStorage.setItem('token', newToken);
    setToken(newToken);
    setIsLoggedIn(true);
  };

  const logout = async () => {
    await AsyncStorage.removeItem('token');
    setToken(null);
    setIsLoggedIn(false);
  };
};
```

6.7 Widget Android

La aplicación incluye un widget nativo para Android que muestra el estado de un sistema seleccionado:

```
// widgets/SystemStatusWidget.tsx
interface SystemStatusWidgetProps {
  systemName: string;
  status: 'UP' | 'DOWN' | 'PENDING' | 'UNKNOWN';
  lastChecked: string;
  responseTime: number | null;
}

export function SystemStatusWidget({
  systemName,
  status,
  lastChecked,
  responseTime,
}: SystemStatusWidgetProps) {
  return (
    <FlexWidget
      style={{
        backgroundColor: '#1E293B',
        borderRadius: 16,
        padding: 16,
      }}
      clickAction="OPEN_APP"
    >
      <TextWidget text={systemName} />
      <TextWidget
        text={getStatusText(status)}
        style={{ color: getStatusColor(status) }}
      />
      <TextWidget text={`Último check: ${lastChecked}`} />
      {responseTime && (
        <TextWidget text={`${responseTime}ms`} />
      )}
    </FlexWidget>
  );
}
```

6.7.1 Características del Widget

- **Tamaño:** 2x1 celdas mínimo
- **Actualización:** Periódica o por evento
- **Acción:** Abre la app al tocar
- **Tema:** Oscuro para mejor visibilidad

6.8 Tema y Colores

Paleta de colores consistente con la versión web:

```
// theme/colors.ts
export const colors = {
  // Backgrounds
  background: '#0F172A',
  backgroundCard: '#1E293B',
  backgroundInput: '#334155',

  // Text
  textMain: '#F8FAFC',
  textMuted: '#94A3B8',

  // Status
  statusSuccess: '#22C55E', // UP - Verde
  statusDanger: '#EF4444', // DOWN - Rojo
  statusWarning: '#F59E0B', // DEGRADED - Amarillo
  statusInfo: '#3B82F6', // Primary - Azul

  // UI
  primary: '#3B82F6',
  border: '#334155',
};
```

6.9 Configuración de Expo

```
// app.json
{
  "expo": {
    "name": "UpTrack",
    "slug": "up-track-mobile",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "splash": {
      "image": "./assets/splash-icon.png",
      "backgroundColor": "#0F172A"
    },
    "ios": {
      "supportsTablet": true,
      "bundleIdentifier": "com.uptrack.mobile"
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/adaptive-icon.png",
        "backgroundColor": "#0F172A"
      }
    }
  }
}
```

```
    },  
    "package": "com.uptrack.mobile"  
  }  
}  
}
```

6.10 Desarrollo Local

6.10.1 Requisitos

- Node.js 18+
- pnpm (recomendado)
- Expo CLI
- Android Studio / Xcode (para emuladores)
- Expo Go app (para dispositivo físico)

6.10.2 Instalación y Ejecución

```
# Instalar dependencias  
cd up-track-mobile  
pnpm install  
  
# Iniciar servidor de desarrollo  
pnpm start  
  
# Ejecutar en Android  
pnpm android  
  
# Ejecutar en iOS  
pnpm ios  
  
# Ejecutar en navegador web  
pnpm web
```

6.10.3 Configuración del API

Editar src/api/config.ts:

```
// Para desarrollo local con dispositivo físico  
export const API_BASE_URL = 'http://192.168.1.X:8080';  
  
// Para emulador Android  
export const API_BASE_URL = 'http://10.0.2.2:8080';  
  
// Para producción  
export const API_BASE_URL = 'https://api.uptrack.example.com';
```

6.11 Build de Producción

6.11.1 EAS Build (Recomendado)

```
# Instalar EAS CLI
npm install -g eas-cli

# Login en Expo
eas login

# Build para Android (APK)
eas build --platform android --profile preview

# Build para iOS
eas build --platform ios --profile preview

# Build de producción
eas build --platform all --profile production
```

6.11.2 Configuración EAS (eas.json)

```
{
  "build": {
    "preview": {
      "distribution": "internal",
      "android": {
        "buildType": "apk"
      }
    },
    "production": {
      "android": {
        "buildType": "app-bundle"
      }
    }
  }
}
```

6.12 Funcionalidades Futuras

Funcionalidad	Estado	Descripción
Push Notifications	Planificado	Alertas en tiempo real vía Firebase/APNs
Widget iOS	Planificado	Widget para pantalla de inicio iOS

Funcionalidad	Estado	Descripción
Modo Offline	Planificado	Caché completo con sincronización
Biometría	Planificado	Autenticación con huella/Face ID
Deep Links	Planificado	Abrir target específico desde notificación

7 Modelo de Datos

La persistencia de UpTrackAI se gestiona en PostgreSQL utilizando GORM como ORM. El esquema está diseñado para soportar alta frecuencia de escritura (health checks) y consultas eficientes de historial.

7.1 Diagrama de Clases / Entidades

El siguiente diagrama ilustra las entidades principales del dominio y sus relaciones.

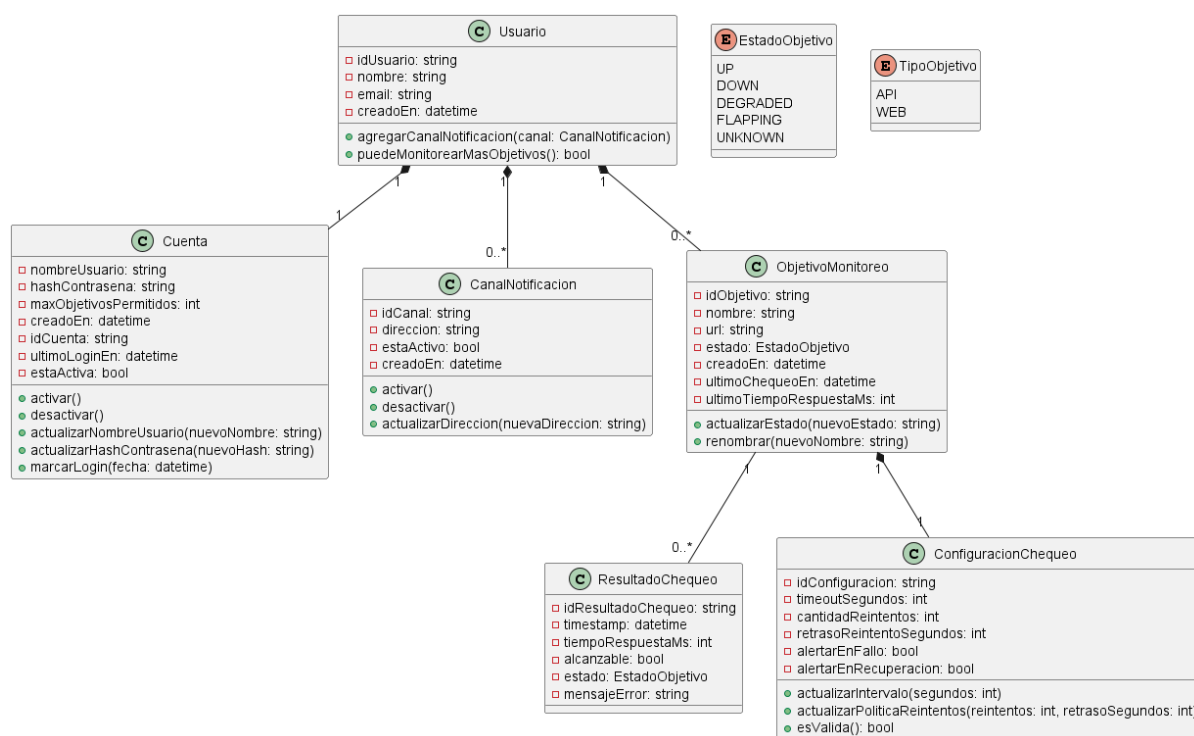


Figure 7.1: Diagrama de Clases

7.2 Tablas del Sistema

7.2.1 users

Almacena información de perfil de los usuarios registrados.

Columna	Tipo	Restricciones	Descripción
id	uuid	PK	Identificador único (UUID v7)
email	varchar(254)	UNIQUE, NOT NULL	Correo electrónico

Columna	Tipo	Restricciones	Descripción
full_name	varchar(255)	-	Nombre completo
avatar_url	varchar(500)	-	URL de avatar
timezone	varchar(50)	DEFAULT 'UTC'	Zona horaria
language	varchar(10)	DEFAULT 'en'	Idioma preferido
created_at	timestamp	AUTO	Fecha de creación
updated_at	timestamp	AUTO	Última actualización

7.2.2 credentials

Almacena credenciales de autenticación separadas del perfil (seguridad).

Columna	Tipo	Restricciones	Descripción
id	uuid	PK	Identificador único
user_id	uuid	FK, UNIQUE	Referencia al usuario
password_hash	varchar(255)	NOT NULL	Hash bcrypt de la contraseña
last_login_at	timestamp	-	Último inicio de sesión
login_attempts	int	DEFAULT 0	Intentos fallidos consecutivos
locked_until	timestamp	-	Bloqueo temporal por intentos
password_changed_at	timestamp	NOT NULL	Última cambio de contraseña
requires_password_change	boolean	DEFAULT false	Forzar cambio en próximo login

7.2.3 monitoring_targets

Tabla principal de objetivos de monitoreo.

Columna	Tipo	Restricciones	Descripción
id	uuid	PK	Identificador único (UUID v7)
user_id	uuid	FK, NOT NULL	Propietario del target
name	varchar(255)	NOT NULL	Nombre descriptivo
url	text	NOT NULL	URL a monitorear
target_type	varchar(50)	NOT NULL	Tipo: 'API' o 'WEB'
is_active	boolean	DEFAULT true	¿Monitoreo activo?
previous_status	varchar(50)	DEFAULT 'UNKNOWN'	Estado anterior (para transiciones)
current_status	varchar(50)	DEFAULT 'UNKNOWN'	Estado actual
check_interval_seconds	int	DEFAULT 300	Frecuencia de verificación (5 min)
timeout_seconds	int	DEFAULT 10	Timeout de conexión
retry_count	int	DEFAULT 3	Reintentos antes de fallar
retry_delay_seconds	int	DEFAULT 1	Pausa entre reintentos

Columna	Tipo	Restricciones	Descripción
last_checked_at	timestamp	-	Última verificación
next_check_at	timestamp	INDEX	Próxima verificación programada

Índices:

- idx_monitoring_targets_next_check_at en next_check_at (polling eficiente)

7.2.4 check_results

Almacena resultados de verificaciones con cambios de estado (alertas).

Columna	Tipo	Restricciones	Descripción
id	uuid	PK	Identificador único
monitoring_target_id	uuid	FK, NOT NULL	Target verificado
timestamp	timestamp	NOT NULL	Momento de la verificación
status	varchar(50)	NOT NULL	Estado determinado
avg_response_time_ms	int	NOT NULL	Tiempo de respuesta promedio
error_message	text	-	Mensaje de error (si aplica)

Índices:

- idx_target_timestamp en (monitoring_target_id, timestamp) compuesto

7.2.5 metrics

Serie temporal de métricas de rendimiento (solo checks exitosos).

Columna	Tipo	Restricciones	Descripción
monitoring_target_id	uuid	FK, NOT NULL	Target monitoreado
timestamp	timestamp	NOT NULL	Momento de la medición
response_time_ms	int	NOT NULL	Latencia en milisegundos

Índices:

- idx_metric_target_time en (monitoring_target_id, timestamp) compuesto

7.2.6 target_statistics

Estadísticas agregadas calculadas para cada target.

Columna	Tipo	Restricciones	Descripción
target_id	uuid	PK, FK	Referencia al target
avg_response_time_ms	int	DEFAULT 0	Promedio de latencia histórico
total_checks_count	int	DEFAULT 0	Total de verificaciones
last_updated_at	timestamp	AUTO	Última actualización

7.2.7 notification_channels

Canales de notificación configurados por usuarios.

Columna	Tipo	Restricciones	Descripción
id	varchar(100)	PK	ID del canal (ej: Telegram Chat ID)
user_id	varchar(36)	FK, NOT NULL, INDEX	Propietario
type	varchar(20)	NOT NULL	Tipo: 'telegram', 'email', etc.
value	text	NOT NULL	Configuración (JSON o string)
priority	int	DEFAULT 0	Prioridad de envío
is_active	boolean	DEFAULT true	¿Canal activo?

7.2.8 telegram_linking_tokens

Tokens temporales para vinculación de cuentas Telegram (Magic Link).

Columna	Tipo	Restricciones	Descripción
token	varchar(64)	PK	Token único generado
user_id	varchar(36)	FK, NOT NULL, INDEX	Usuario solicitante
expires_at	timestamp	NOT NULL, INDEX	Fecha de expiración (15 min)
used	boolean	DEFAULT false	¿Token ya utilizado?

7.3 Migraciones

Las migraciones se ejecutan automáticamente al iniciar la aplicación mediante GORM AutoMigrate:

```
// config/migrations.go
func RunMigrations(db *gorm.DB) {
    db.AutoMigrate(
        // User module
        &userpostgres.UserEntity{},

        // Security module
        &securitypostgres.CredentialEntity{},

        // Monitoring module
    )
}
```



```

    &monitoringpostgres.MonitoringTargetEntity{},
    &monitoringpostgres.CheckResultEntity{},
    &monitoringpostgres.MetricEntity{},
    &monitoringpostgres.TargetStatisticsEntity{},

    // Notifications module
    &notifpostgres.NotificationChannelEntity{},
    &notifpostgres.TelegramLinkingToken{},
)
}

```

Características:

- Creación automática de tablas si no existen
- Adición de columnas nuevas
- No elimina columnas existentes (seguro)
- No modifica tipos de columnas existentes

7.4 Consideraciones de Diseño

7.4.1 Separación de Datos Calientes/Fríos

Tabla	Tipo	Frecuencia	Retención
metrics	Caliente	Alta escritura (cada check)	7-30 días
check_results	Templada	Solo cambios de estado	90 días
target_statistics	Fría	Actualización periódica	Indefinida

7.4.2 Estrategia de Índices

```

-- Consultas frecuentes optimizadas:

-- 1. Polling del scheduler (cada 10s)
SELECT * FROM monitoring_targets
WHERE is_active = true
      AND (next_check_at <= NOW() OR next_check_at IS NULL);
-- Índice: idx_monitoring_targets_next_check_at

-- 2. Historial de un target (dashboard)
SELECT * FROM check_results
WHERE monitoring_target_id = ?
ORDER BY timestamp DESC
LIMIT 50;
-- Índice compuesto: idx_target_timestamp

-- 3. Métricas para gráficos
SELECT * FROM metrics

```

```
WHERE monitoring_target_id = ?
AND timestamp > NOW() - INTERVAL '7 days'
ORDER BY timestamp;
-- Índice compuesto: idx_metric_target_time
```

7.4.3 UUID v7 como Primary Key

Se utiliza UUID v7 en lugar de auto-increment por:

- **Ordenamiento temporal:** UUID v7 incluye timestamp, preserva orden de inserción
- **Distribución:** No hay colisiones en sistemas distribuidos
- **Seguridad:** No expone información sobre cantidad de registros

```
// Generación en GORM hooks
func (e *Entity) BeforeCreate(tx *gorm.DB) error {
    if e.ID == uuid.Nil {
        e.ID = uuid.Must(uuid.NewV7())
    }
    return nil
}
```

7.5 Conexión y Pool

La conexión se configura mediante variables de entorno:

```
// config/database.go
dsn := fmt.Sprintf(
    "host=%s port=%s user=%s password=%s dbname=%s sslmode=%s",
    os.Getenv("DB_HOST"),
    os.Getenv("DB_PORT"),
    os.Getenv("DB_USER"),
    os.Getenv("DB_PASSWORD"),
    os.Getenv("DB_NAME"),
    os.Getenv("DB_SSLMODE"),
)

db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
```

Configuración del Pool (recomendada para producción):

```
sqlDB, _ := db.DB()
sqlDB.SetMaxIdleConns(10)
sqlDB.SetMaxOpenConns(100)
sqlDB.SetConnMaxLifetime(time.Hour)
```

7.6 Backup y Recuperación

7.6.1 Estrategia Recomendada

Tipo	Frecuencia	Herramienta	Retención
Full Backup	Diario	pg_dump	7 días
WAL Archiving	Continuo	pg_basebackup	24 horas
Snapshot	Semanal	Cloud Provider	30 días

7.6.2 Comandos de Backup

```
# Backup completo
pg_dump -h localhost -U postgres -d uptrackai_db -F c -f backup.dump

# Restauración
pg_restore -h localhost -U postgres -d uptrackai_db -c backup.dump

# Backup solo datos (sin esquema)
pg_dump -h localhost -U postgres -d uptrackai_db --data-only > data.sql
```

7.7 Limpieza de Datos

Script recomendado para mantenimiento (ejecutar semanalmente):

```
-- Eliminar métricas antiguas (> 30 días)
DELETE FROM metrics
WHERE timestamp < NOW() - INTERVAL '30 days';

-- Eliminar tokens de linking expirados
DELETE FROM telegram_linking_tokens
WHERE expires_at < NOW();

-- Vacuum para recuperar espacio
VACUUM ANALYZE metrics;
VACUUM ANALYZE check_results;
```